CSCE 629-601 Analysis of Algorithms

Fall 2022

Instructor: Dr. Jianer Chen
Office: PETR 428
Phone: (979) 845-4259
Email: chen@cse.tamu.edu
Office Hours: MWF 3:50pm–5:00pm

Teaching Assistant: Vaibhav Bajaj
Office: EABC 107B
Phone: (979) 739-2707
Email: vaibhavbajaj@tamu.edu
Office Hours: T; 2pm-3pm, TR: 4pm-5pm

Course Notes #4. Hashing

1 Preliminaries

Let U be a universal set of elements that are of interest to our applications. Let $m \geq 0$ be a fixed integer. We call a function $h : U \to [0..m-1]$ a *hash function* from U to $[0..m-1]$.

Each hash function h gives a hashing structure, as follows: we prepare an array $H_h[0..m-1]$ of doubly linked lists such that an element x in U will be "hashed" into the linked list $H_h[h(x)]$. Therefore, the SEARCH, INSERT, and DELETE operations based on the hashing structure H_h can be implemented as follows:

- SEARCH(x): search the element $x \in U$ in the list $H_h[h(x)]$.
- INSERT(x): insert the element $x \in U$ into the head of the list $H_h[h(x)]$.
- DELETE(p_x): delete the node pointed by the pointer p_x in the structure H_h.

The operation INSERT(x) takes time $O(1)$ because we always insert the new node into the head of the list $H_h[h(x)]$. The operation DELETE(p_x) also takes time $O(1)$ since the node to be deleted can be accessed directly by the pointer p_x and all lists in the structure H_h are doubly linked (also note that the operation DELETE(p_x) does not even have to know the element x). Thus, the only non-trivial operation is SEARCH(x), whose running time is $O(d+1)$, where d is the number of nodes before the node containing the element x in the list $H_h[h(x)]$ (in caes x is not in H_h, d is the total number of nodes in the list $H_h[h(x)]$).

Note that there are other possible versions of INSERT and DELETE. For example, the operation INSERT(x) may require to either insert x when x does not exist in the current structure H_h or simply report the existence of x in H_h. Similarly, DELETE may have the form DELETE(x), where $x \in U$, which either deletes the node containing the element x in the list $H_h[h(x)]$ or reports the nonexistence of x in H_h. Each of these extensions is equivalent to a SEARCH(x) operation followed possibly by an INSERT/DELETE operation given in the above list. Thus, we only need to focus on the three "basic" operations listed above, and the complexity of the extended versions of the operations will follow easily.

2 Complexity based on ideal hash functions

We say that a hash function h is *ideal* if it hashes each element x of U into each integer i, $0 \leq i \leq m-1$, with a probability $1/m$.[1]

[1]Perhaps the name "perfect hash function" sounds more proper. However, "perfect hash function" has its formal definition that is different from what we wanted to mean here and has become standard in the hashing

Since both INSERT and DELETE take time $O(1)$, we only need to consider the operation SEARCH(x). In this section, we study the complexity of SEARCH based on the assumption that we have an ideal hash function h. Let $X_n = \{x_1, x_2, \ldots, x_n\}$ be a subset of n elements in U that have been placed in the structure H_h, in that order. There are two difference cases.

Case 1. The element x is not in H_h.

In this case, we need to search the entire list $H_h[h(x)]$. Thus, the running time of SEARCH(x) is $O(d+1)$, where d is the number of nodes in the list $H_h[h(x)]$ (the number 1 in the expression makes the case when the list $H_h[h(x)]$ is empty meaningful). The worst case can be very bad: if all elements in X_n are in the list $H_h[h(x)]$, then the search time will be $O(n)$. Note that this situation can happen even we assume that the hash function h is ideal. For example, if $|U| > nm$, then there must be an index k such that more than n elements in U are hashed into $H_h[k]$. Therefore, the worst situation occurs if X_n happens to be such n elements and $h(x) = k$.

We are more interested in the "average" length of the list $H_h[k]$ for a general index k. For this, we define for each element x_i in X_n, $1 \le i \le n$, the following random variable:

$$Y_{k,i} = \begin{cases} 1 & \text{if } h(x_i) = k \\ 0 & \text{if } h(x_i) \neq k \end{cases}$$

Based on this, we can define a random variable $Y(k)$ for the length of the list $H_h[k]$ as $Y(k) = \sum_{i=1}^{n} Y_{k,i}$.

Lemma 1 *For each fixed k, $0 \le k \le m-1$, the expected length of the list $H_h[k]$ is n/m.*

PROOF. Since the hash function h is ideal, for a fixed k, we have $E[Y_{k,i}] = \Pr[Y_{k,i} = 1] = 1/m$ for each element x_i. By the linearality of the expectation, we have

$$E[Y(k)] = E\left[\sum_{i=1}^{n} Y_{k,i}\right] = \sum_{i=1}^{n} E[Y_{k,i}] = n/m.$$

Thus, the expected value for the random variable $Y(k)$ is n/m, which proves the lemma. □

In case x is not in H_h, the operation SEARCH(x) takes time $O(d+1)$, where d is the number of nodes in the list $H_h[k]$, assuming $h(x) = k$. By Lemma 1, the expected running time of the operation SEARCH(x) is $O(n/m + 1)$.

Case 2. The element x is in H_h.

Now we suppose that the element x under the search is in the hashing structure H_h. Since for different x in H_h, the search time on x may be different, we make another assumption that x is any one of the elements $\{x_1, x_2, \ldots, x_n\}$ in H_h with an equal probability, i.e., for each i, $\Pr[x = x_i] = 1/n$. Now for a fixed i, suppose $x = x_i$. Then the running time of SEARCH(x) is $O(d+1)$, where d is the total number of nodes that are before x_i in the list $H_h[k]$, where $k = h(x_i)$. Consider the following random variables for $1 \le j \le n$:

$$Z_{i,j} = \begin{cases} 1 & \text{if } x = x_i,\ h(x_j) = h(x),\ j > i \\ 0 & \text{otherwise} \end{cases}$$

literature. Thus, we adopt to use "ideal hash function" instead.

Note that for each $j \neq i$,

$$\Pr[x = x_i, h(x_j) = h(x)] = \Pr[h(x_j) = h(x) \mid x = x_i] \cdot \Pr[x = x_i] = 1/(nm),$$

where the equality $\Pr[h(x_i) = h(x_j) \mid x = x_i] = 1/m$ is because on the condition $x = x_i$, $h(x_j)$ is equal to the value $h(x_i)$ $(= h(x))$ with a probability $1/m$. Therefore, for each j, we have

$$E[Z_{i,j}] = \begin{cases} 1/(nm) & \text{if } j > i \\ 0 & \text{otherwise} \end{cases}$$

The condition $\{x = x_i, h(x_j) = h(x), j > i\}$ give exactly the condition that x_j is a node before x_i in the list $H_h[h(x)]$. Thus, when $x = x_i$, $Z_i = \sum_{j=i+1}^{n} Z_{i,j}$ is the number of nodes before x in the list $H_h[h(x)]$. Since $Z_{i,j} = 0$ for all j if $x \neq x_i$, we have $Z_i = 0$ if $x \neq x_i$. Therefore, if x is in H_h, $Z = \sum_{i=1}^{n} Z_i$ would be exactly the number of nodes before x in the linked list that contains x. The expected value of Z can be easily computed using the linearality of expections:

$$\begin{aligned} E[Z] &= E\left[\sum_{i=1}^{n} Z_i\right] = \sum_{i=1}^{n} E[Z_i] = \sum_{i=1}^{n} E\left[\sum_{j=i+1}^{n} Z_{i,j}\right] = \sum_{i=1}^{n}\sum_{j=i+1}^{n} E[Z_{i,j}] \\ &= \sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{1}{nm} = \frac{1}{nm}\sum_{i=1}^{n}(n-i) = \frac{n-1}{2m} < \frac{n}{2m}. \end{aligned}$$

In summary, if x is in the hash structure H_h with an equal probability to be each of the elements in H_h, then SEARCH(x) takes expected time $O(1 + n/(2m))$.

3 Universal hashing

Fix a set U and an integer $m \geq 0$. Let $\mathcal{H}$ be a collection of hash functions from U to $[0, , m-1]$. We say that $\mathcal{H}$ is a *universal hashing* if for any two elements x and y in U, there are at most $|\mathcal{H}|/m$ hash functions h that make $h(x) = h(y)$. In other words, if we randomly pick a hash function h from $\mathcal{H}$, then the probability that $h(x) = h(y)$ is bounded by $1/m$.

The discussion in this section requires some familiarity with Number Theory. We will try to give a presentation that requires the minimum background in Number Theory. Let p be a prime number. For any integer a, denote by $a \pmod p$ the remainder of a divided by p, and write $a \equiv b \pmod p$ if $|a - b|$ is divisible by p. The numbers in $[0..p-1]$ form a *field* in which we can apply the operations $+$, $-$, $\times$, and $/$ as normal but "mod p". In particular, if $a \equiv b \pmod p$ and $c \equiv d \pmod p$, then $(a - c) \equiv (b - d) \pmod p$.

3.1 Constructing universal hashing

We show how to construct a universal hashing given U and m. The elements in the set U can be given, without loss of generality, as the integers in $[0..|U| - 1]$. Let p be a prime that is not smaller than $|U|$. Define a collection of hash functions from U to $[0..m-1]$ as

$$\mathcal{H}_{U,m} = \{h_{a,b,m} \mid 1 \leq a \leq p-1, 0 \leq b \leq p-1\}, \tag{1}$$

where $h_{a,b,m}$ is defined as $h_{a,b,m}(x) = ((ax + b) \bmod p) \bmod m$.

Fix the two elements x and y in U, with $x \neq y$. Consider the following two sets:

$$S_1 = \{(a,b) \mid 1 \leq a \leq p-1, 0 \leq b \leq p-1\} \quad \text{and} \quad S_2 = \{(s,t) \mid 0 \leq s,t \leq p-1, s \neq t\}.$$

The two elements x and y induce a mapping π from S_1 to S_2 by

$$\pi \text{ from } (a,b) \in S_1 \text{ to } (s,t) \in S_2: \qquad s = (ax+b) \bmod p \quad \text{and} \quad t = (ay+b) \bmod p.$$

Note that the condition $s \neq t$ is ensured by the following fact: if $s = t$, then $a(x-y) \pmod p = 0$, which, with $0 < |x-y| < p$, would imply that the prime p is divisible by a with $2 \leq a \leq p-1$.

The mapping π from S_1 to S_2 is actually surjective, i.e., different pairs in S_1 are mapped to different pairs in S_2. To see this, suppose the contrary that there are two different pairs (a_1, b_1) and (a_2, b_2) in S_1 such that

$$(a_1x + b_1) \equiv (a_2x + b_2) \pmod p \quad \text{and} \quad (a_1y + b_1) \equiv (a_2y + b_2) \pmod p.$$

Subtracting the second from the first, we get

$$a_1(x-y) \equiv a_2(x-y) \pmod p \quad \text{i.e.,} \quad (a_1 - a_2)(x-y) \equiv 0 \pmod p.$$

Since $0 < |x-y| < p$, this implies $a_1 = a_2$. Combining this with $(a_1x+b_1) \equiv (a_2x+b_2) \pmod p$ gives $b_1 = b_2$, contradicting the assumption that (a_1, b_1) and (a_2, b_2) are different pairs in S_1.

Note that the number of pairs in S_1 is equal to the number of pairs in S_2. Thus, the mapping π is actually a one-to-one mapping between S_1 and S_2. Therefore, randomly picking a pair (a,b) in S_1 and making a pair (s,t) in S_2, where $s = (ax+b) \pmod p$ and $t = (ay+b) \pmod p$, can be regarded as a process of randomly picking a pair (s,t) in S_2.

By the definition of the hashing function, $h_{a,b,m}(x) = s \pmod m$ and $h_{a,b,m}(y) = t \pmod m$.

We count the number of pairs (s,t) in S_2 such that $s \equiv t \pmod m$. For each s with $0 \leq s \leq p-1$, there are $\lceil p/m \rceil - 1 \leq (p-1)/m$ numbers t such that $0 \leq t \leq p-1$, $t \neq s$, and $s \equiv t \pmod m$. Therefore, the total number of pairs (s,t) in S_2 with $s \equiv t \pmod m$ is bounded by $p(p-1)/m$. Since there is a one-to-one mapping π from the set S_1 to the set S_2, the total number of pairs (a,b) in S_1 such that $\pi(a,b) = (s,t)$ with $s \equiv t \pmod m$ is bounded by $p(p-1)/m$. Note that $\pi(a,b) = (s,t)$ implies $s = (ax+b) \pmod p$, $t = (ay+b) \pmod p$, and that $s \pmod m = h_{a,b,m}(x)$, $t \pmod m = h_{a,b,m}(y)$. Therefore, the above conclusion implies that the number of pairs (a,b) in S_1 that make $h_{a,b,m}(x) = h_{a,b,m}(y)$ is bounded by $p(p-1)/m$. Since there are totally $p(p-1)$ pairs in S_1, which correspond to $p(p-1)$ hash functions $h_{a,b,m}$, where $(a,b) \in S_1$, we conclude that if we randomly pick a pair (a,b) in S_1 and make a hash function $h_{a,b,m}$, then the probability that $h_{a,b,m}(x) = h_{a,b,m}(y)$ is bounded by $(p(p-1)/m)/(p(p-1)) = 1/m$.

Therefore, for the two elements x and y in U, with $x \neq y$, if we randomly pick a pair (a,b) in S_1 and use the hash function $h_{a,b,m}$, then the probability that $h_{a,b,m}(x) = h_{a,b,m}(y)$ is bounded by $1/m$. This proves that the collection in (1) is a universal hashing from the set U to $[0..m-1]$.

3.2 How good is a hash function randomly picked from universal hashing?

Intuitively, a hash function h randomly picked from a universal hashing $\mathcal{H}_{U,m}$ is good: for any two different elements in the set U, the function h causes a collision with a probability bounded by $1/m$. However, this does not imply directly that the same hash function h is good for *all* pairs of different elements in U. In this subsection, we show that the same hash function h randomly picked from the universal hashing $\mathcal{H}_{U,m}$ can be expected to be always good.

As we have seen early, the most time-consuming operation for hashing is SEARCH(x), which is proportional to the length of the linked list $H[h(x)]$. For this, we assume that we have used the hash function h to distribute all elements of $U = \{x_1, \ldots, x_n\}$ in the hash table $H[0..m-1]$, and that we will next to do a SEARCH(x), where $h(x) = k$.

Case 1. x is not in U.

For each element x_i in U, since h is randomly picked from $\mathcal{H}_{U,m}$, $\text{Prob}[h(x_i) = h(x)] \leq 1/m$. We define a random variable as follows:

$$Z_{x,x_i} = \begin{cases} 1 & \text{if } h(x_i) = h(x) \\ 0 & \text{otherwise} \end{cases}$$

From this, we have $\text{Exp}[Z_{x,x_i}] = \text{Prob}[h(x_i) = h(x)] \leq 1/m$.

The length of the list $H[k]$ is $L_x = \sum_{i=1}^{n} Z_{x,x_i}$, and the expected value of L_x is

$$\text{Exp}[L_x] = \text{Exp}\left[\sum_{i=1}^{n} Z_{x,x_i}\right] = \sum_{i=1}^{n} \text{Exp}[Z_{x,x_i}] \leq \sum_{i=1}^{n} (1/m) = n/m.$$

In conclusion, if x is not in the hash table $H[0..m-1]$, then the expected time for the operation SEARCH(x) is bounded by $O(n/m)$.

Case 2. x is in U.

Using the above notations, in this case, the number of elements in U that are in $H[k]$ but are not x is equal to $L'_x = \sum_{x_i \neq x} Z_{x,x_i}$, whose expected value is

$$\text{Exp}[L'_x] = \text{Exp}\left[\sum_{x_i \neq x} Z_{x,x_i}\right] = \sum_{x_i \neq x} \text{Exp}[Z_{x,x_i}] \leq \sum_{x_i \neq x} (1/m) = (n-1)/m < n/m.$$

Thus, again the expected time for the operation SEARCH(x) is bounded by $O(n/m)$.

In summary, in all cases, for any given set U of n elements, if we randomly pick a hash function h in the universal hashing $\mathcal{H}_{U,m}$ (note that $\mathcal{H}_{U,m}$ depends only on the number of elements in U but not on the actual set U), then the constructed hash table supports the SEARCH operation in expected time $O(n/m)$, which is the best possible we can hope.

4 Perfect hashing

So far, our evaluation of hash functions is based on the "expected performance," i.e., the performance of the function in most cases. However, this does not exclude the possibility that in some extreme cases (i.e., cases that happen with small probability), the function performs very badly. An interesting question is whether there are hash functions that always perform well, even in the *worst* case. If there are such hash functions, how do we construct them, and how expensive is the construction? We will study this issue in the current section.

Let A be a set of n elements in the universal set U. A hash function h from the set A to $[0..m-1]$ is *perfect* if no two elements x and y in A cause a collision $h(x) = h(y)$. Of course, in this case, we must have $n \leq m$.

Let p be a prime that is larger than $|U|$. Recall that the following collection

$$\mathcal{H}_{U,m} = \{h_{a,b,m} \mid 1 \leq a \leq p-1, 0 \leq b \leq p-1\}$$

where $h_{a,b,m}(x) = ((ax + b) \bmod p) \bmod m$, makes a universal hashing from U to $[0..m-1]$.

We first show that if we allow m to be sufficiently large, then a perfect hashing from A to $[0..m-1]$ can be easily constructed, based on the following lemma.

Lemma 2 *For a given set A of n elements, if we randomly pick a hash function h from $\mathcal{H}_{U,n^2}$, then the probability that h is a perfect hash function from A to $[0..n^2-1]$ is larger than $1/2$.*

PROOF. Let x and y be two fixed different elements in A. By the definition of universal hashing $\mathcal{H}_{U,n^2}$, if we randomly pick a hash function h from $\mathcal{H}_{U,n^2}$, then the probability that $h(x) = h(y)$ is bounded by $1/n^2$. Since there are $\binom{n}{2}$ pairs of different elements in A, the probability that $h(x) = h(y)$ for *any* two different elements x and y in A is bounded by

$$\sum_{x,y \in A,\, x \neq y} \text{Prob}[h(x) = h(y)] \leq \sum_{x,y \in A,\, x \neq y} \frac{1}{n^2} = \binom{n}{2} \cdot \frac{1}{n^2} = \frac{n(n-1)}{2} \cdot \frac{1}{n^2} < \frac{1}{2}.$$

Therefore, the probability that no two different elements x and y make $h(x) = h(y)$, i.e., the probability that the function h is perfect, is larger than $1/2$. □

In particular, Lemma 2 implies that there are (more than one half of the) hash functions in $\mathcal{H}_{U,n^2}$ that are perfect from A to $[0..n^2-1]$. Lemma 2 also suggests a randomized algorithm to find such a perfect hash function: simply repeatedly pick randomly a hash function in $\mathcal{H}_{U,n^2}$ until a perfect hash function is found. Note that randomly picking a hash function in $\mathcal{H}_{U,n^2}$ can be implemented using the techniques described in Section 3, and that checking whether a hash function is perfect is straightforward. By Lemma 2, with a large probability, this randomized algorithm will find a perfect hash function very quickly.

The drawback of the above construction is $m = n^2$ is too large, which makes the hash structure very space expensive. In the following, we study how to reduce the size of the hash table. For this, we first consider hash functions for the set A whose hash table has a size n.

Lemma 3 *Suppose that we randomly pick a hash function h from the universal hashing $\mathcal{H}_{U,n}$, and let n_i be the number of elements in A that are hashed into the list $H[i]$, for $0 \leq i \leq n-1$. Then $\text{Exp}[\sum_{i=0}^{n-1} n_i^2] < 2n$.*

PROOF. First note that

$$n_i^2 = 2 \cdot \frac{n_i^2}{2} = 2\left(\frac{n_i(n_i-1)}{2} + \frac{n_i}{2}\right) = 2\left(\binom{n_i}{2} + \frac{n_i}{2}\right) = 2 \cdot \binom{n_i}{2} + n_i.$$

Therefore,

$$\text{Exp}\left[\sum_{i=0}^{n-1} n_i^2\right] = \text{Exp}\left[\sum_{i=0}^{n-1} \left(2 \cdot \binom{n_i}{2} + n_i\right)\right] = 2 \cdot \text{Exp}\left[\sum_{i=0}^{n-1} \binom{n_i}{2}\right] + \text{Exp}\left[\sum_{i=0}^{n-1} n_i\right]$$

The sum $\sum_{i=0}^{n-1} n_i$ is always equal to n, so $\text{Exp}[\sum_{i=0}^{n-1} n_i] = n$. Two elements x and y collide by h, i.e., $h(x) = h(y)$ if and only if x and y are hashed into the same list $H[i]$ for some i. Therefore, $\sum_{i=0}^{n-1} \binom{n_i}{2}$ is exactly the number of pairs of elements in A that collide under h. Since the hash function h was picked randomly from the universal hashing $\mathcal{H}_{U,n}$, by the definition of universal hashing, two elements in A collide with a probability bounded by $1/n$. Now there are $\binom{n}{2}$ pairs in A, so the expected number of collisions under h is bounded by $\binom{n}{2} \cdot (1/n) = (n-1)/2$, i.e., $\text{Exp}[\sum_{i=0}^{n-1} \binom{n_i}{2}] \leq (n-1)/2$. Summarizing all these, we get $\text{Exp}[\sum_{i=0}^{n-1} n_i^2] \leq 2n - 1 < 2n$. □

Lemma 3 implies that there are hash functions in $\mathcal{H}_{U,n}$ that satisfy $\sum_{i=0}^{n-1} n_i^2 < 2n$. Moreover, based on the study in probability theory (Markov Inequality), there are many hash functions in $\mathcal{H}_{U,n}$ that satisfy, say, $\sum_{i=0}^{n-1} n_i^2 < 3n$, which can be found quickly with a large probability by repeatedly picking a random hash function in $\mathcal{H}_{U,n}$. Note that hash functions satisfying $\sum_{i=0}^{n-1} n_i^2 < 3n$ will also meet our need in the construction of perfect hash functions, with a slightly larger space expense.

Now we are ready to present our algorithm that constructs a perfect hash function from the set A of n elements to $[0..N-1]$, where $N < 2n$. The algorithm is given below.

Algorithm PerfectHash(A)

1. pick a prime $p > |U|$;
2. pick a and b with $1 \le a \le p-1$, $0 \le b \le p-1$ such that the hash function $h_{a,b,n}(x) = ((ax+b) \bmod p) \bmod n$ satisfies $\sum_{i=0}^{n-1} |X_i|^2 < 2n$, where for all $0 \le i \le n-1$, X_i is the set of elements x in A such that $h(x) = i$;
3. **for** $i = 0$ **to** $n-1$ **do**
   pick a_i and b_i with $1 \le a_i \le p-1$, $0 \le b_i \le p-1$ such that the hash function $h_{a_i,b_i,|X_i|^2}(x) = ((a_i x + b_i) \bmod p) \bmod |X_i|^2$ is perfect from X_i to $[0..|X_i|^2 - 1]$;
4. output $\{(a,b),(a_0,b_0),\ldots,(a_{n-1},b_{n-1})\}$.

The existence of the numbers a and b in step 2 of the algorithm is ensured by Lemma 3. The existence of the numbers a_i and b_i in step 3, for $0 \le i \le n-1$, is ensured by Lemma 2. The result of the algorithm is a collection of $n+1$ pairs $\{(a,b),(a_0,b_0),\ldots,(a_{n-1},b_{n-1})\}$. The hash table is $H_A[0..N-1]$, where $N = \sum_{i=0}^{n-1} |X_i|^2 < 2n$, which is the concatenation of the hash tables from X_i to $[0..|X_i|^2 - 1]$ for $0 \le i \le n-1$ constructed in step 3. In order to make the hashing function constant-time computable, we will pre-store: (1) the numbers p and n, (2) the numbers $\{(a,b),(a_0,b_0),\ldots,(a_{n-1},b_{n-1})\}$, and (3) n additional numbers d_0, d_1, ..., d_{n-1}, where $d_i = \sum_{h=0}^{i-1} |X_h|^2$ for $0 \le i \le n-1$. Note that the subtable $H_A[d_i..d_{i+1}-1]$, which is of size $|X_i|^2$, is for the hash table for the set X_i constructed in step 3 of the algorithm. All these subtables are disjoint in $H_A[0..N-1]$.

For a given element x, the hash function h_A for A on the hash table $H_A[0..N-1]$ is computed as follows: (1) compute $i = h_{a,b,n}(x) = ((ax+b) \bmod p) \bmod n$ using the pre-stored numbers a, b, p, and n; (2) compute $k = h_{a_i,b_i,|X_i|^2}(x) = ((a_i x + b_i) \bmod p) \bmod |X_i|^2$ using the pre-stored numbers a_i, b_i, and p, where $|X_i|$ is computed by $|X_i| = d_{i+1} - d_i$; (3) let $h_A(x) = d_i + k$.

Step 1 in the above procedure determines which set X_i the element x belongs to. By our construction, the hash function $h_{a_i,b_i,|X_i|^2}(x)$ is perfect from X_i to $[0..|X_i|^2 - 1]$, which corresponds to the subtable $H_A[d_i..d_{i+1}-1]$ in the hash table $H_A[0..N-1]$. Thus, step 3 of the above procedure places x in the hash table $H_A[0..N-1]$ which will not collide with any other elements in A. In summary, the hash function h_A can be computed in constant time, and is perfect from A to $[0..N-1]$.

A straightforward implementation of the algorithm PerfectHash takes time $O(n^2p^2)$, which is probably not practical (recall that the number p can be very large). On the other hand, as we remarked early, the hash functions $h_{a,b,m}$ and $h_{a_i,b_i,|X_i|^2}$ for $0 \le i \le n-1$ can be constructed by randomly picking the numbers a, b, a_i, b_i in $[0..p-1]$. With a large probability, this randomized process will construct the hash functions in time $O(n^2)$. On the other hand, if we adopt this construction, then the upper bound of the size of the hash table $H_A[0..N-1]$ will be increased somehow, for example from $2n$ to $3n$. We leave the detailed study of this implementation to the interested readers.