CSC265 Homework 5 | Fall 2021
Name: Haojun Qiu | Student Number: 1006900622
Name: Runshi Yang | Student Number: 1006734223

Q1

(a)

Let $\mathcal{H}$ be a 3-universal family of hash functions. And let h be a function $\mathcal{H}$. Let $x_1, x_2, x_3 \in U$ be distinct. Then, first notice that $A = \{h \in \mathcal{H} : h(x_1) = h(x_2)\} = \bigsqcup_{i=0}^{m-1}\{h \in \mathcal{H} : h(x_1) = i, h(x_2) = i\}$. And, $A = \bigsqcup_{k=0}^{m-1}\bigsqcup_{i=0}^{m-1}\{h \in \mathcal{H} : h(x_1) = i, h(x_2) = i, \text{and } h(x_3) = k\}$. Therefore,

$$\begin{aligned}
\text{Prob}_{h\in\mathcal{H}}[h(x_1) = h(x_2)] &= \text{Prob}\, A \\
&= \text{Prob} \bigsqcup_{k=0}^{m-1} \bigsqcup_{i=0}^{m-1} \{h \in \mathcal{H} : h(x_1) = i, h(x_2) = i, \text{and } h(x_3) = k\} \\
&= \sum_{k=0}^{m-1}\sum_{i=0}^{m-1} \text{Prob}\{h \in \mathcal{H} : h(x_1) = i, h(x_2) = i, \text{and } h(x_3) = k\} \\
&\qquad\qquad \text{(since component events are disjoint)} \\
&= \sum_{k=0}^{m-1}\sum_{i=0}^{m-1} \text{Prob}_{h\in\mathcal{H}}[h(x_1) = i, h(x_2) = i, \text{and } h(x_3) = k] \\
&= m \cdot m \cdot \frac{1}{m^3} = \frac{1}{m},
\end{aligned}$$

since h is from a 3-universal hash family, and x_1, x_2, x_3 are distinct elemetn of U with each $i, k \in \{0, \ldots, m-1\}$. This proves that $\mathcal{H}$ is universal.

(b)

- Fix x_1, x_2, x_3 to be three distinct number in $\{0, \ldots, m-1\}$ and fix $y_1, y_2, y_3 \in \{0, \ldots, m-1\}$.
- Since $\mathcal{H} = \{h_{a,b,c} | a, b, c \in U\}$, $|U| = m$, and $h_{a,b,c} = ax^2 + bx + c$, we know that $|\mathcal{H}| = m^3$.
- Thus if we show that there is only one $h_{a,b,c} \in \mathcal{H}$ that satisfies $h_{a,b,c}(x_1) = y_1$, $h_{a,b,c}(x_2) = y_2$, $h_{a,b,c}(x_3) = y_3$, we can conclude that $\text{Prob}_{h_{a,b,c} \in \mathcal{H}}[h_{a,b,c}(x_1) = y_1, h_{a,b,c}(x_2) = y_2, h_{a,b,c}(x_3) = y_3] = \frac{1}{m^3}$.
- Let $a, b, c \in U$ such that

$$\begin{cases} h_{a,b,c}(x_1) = y_1 \\ h_{a,b,c}(x_2) = y_2 \\ h_{a,b,c}(x_3) = y_3 \end{cases}$$

which means

$$\begin{cases} x_1^2 a + x_1 b + c \equiv y_1 (\text{mod } m) \\ x_2^2 a + x_2 b + c \equiv y_2 (\text{mod } m) \\ x_3^2 a + x_3 b + c \equiv y_3 (\text{mod } m) \end{cases}$$

In modular arithmetic, we can add, subtract or multiply a constant on both sides of the equation at the same time. (i.e. the first modular equation is equivalent to $k(x_1^2 a + x_1 b + c) \equiv k y_1 (\text{mod } m)$, $x_1^2 a + x_1 b + c + k \equiv y_1 + k (\text{mod } m)$) And since m is a prime number, $gcd(k, m) = 1$ for all $k \in \mathbb{N}^+$, so we can also find the modular inverse of any x mod m. (i.e. the first modular equation is equivalent to $\frac{x_1^2 a + x_1 b + c}{k} \equiv \frac{y_1}{k} (\text{mod } m)$) So we can treat this system of modular equation like an ordinary system of equation. But the solution to this system should be in mod m field, that is a, b, c should all in $\{0, \ldots, m-1\}$.

Since x_1, x_2 and x_3 are distinct, we know that these three equations are linearly independent, so there must be exactly one solution in the mod m field.

Actually we tried to solve this using Symbolab and there is a very long solution (the value for a, b and c are the values in the last column of the matrix):

$$\begin{pmatrix} 1 & 0 & 0 & \frac{x_1^2 y_1 x_3^2 - x_1^2 y_2 x_3^2 - x_1 y_1 x_3^3 x_3 + x_1 y_2 x_3^3 x_3 - x_1^2 y_1 x_2 x_3 + x_1^2 x_1 y_2 x_3 - x_1^2 x_1 x_3 y_3 + x_1^2 x_2 x_3 y_3 + x_1 y_1 x_2 x_3^3 - x_1^2 y_2 x_3^3 + x_1^2 x_3^3 y_3 - x_1 x_2 x_3^3 y_3}{(x_1 x_3^3 - x_1^2 x_3)(x_1^2 x_2 - x_1^2 x_3 + x_1 x_3^3 + x_2^2 x_3 - x_1 x_2^2 - x_2 x_3^3)} \\ 0 & 1 & 0 & \frac{-x_1^2 y_2 x_3 + x_1^2 x_3 y_3 + x_1^2 x_1 y_2 x_3^3 - x_1^2 y_1 x_3^3 x_3 + x_1^2 y_2 x_3^3 x_3 + x_1^2 y_1 x_2^2 x_3 - x_1^2 x_1 x_3^3 y_3 - x_1^2 x_2^2 x_3 y_3 + x_1 y_1 x_3^3 - x_1 y_2 x_3^3 - x_1 y_1 x_2^2 x_3^3 + x_1 x_2^2 x_3^3 y_3}{(x_1 x_3^3 - x_1^2 x_3)(x_1^2 x_2 - x_1^2 x_3 + x_1 x_3^3 + x_2^2 x_3 - x_1 x_2^2 - x_2 x_3^3)} \\ 0 & 0 & 1 & \frac{-y_1 x_2 x_3^3 + x_1 y_2 x_3^3 - x_1^2 y_2 x_3 + y_1 x_2^2 x_3 - x_1 x_2^2 y_3 + x_1^2 x_2 y_3}{x_1 x_3^3 - x_2 x_3^3 - x_1 x_2^2 + x_1^2 x_2 - x_1^2 x_3 + x_2^2 x_3} \end{pmatrix}$$

Since we have shown that there is a unique solution for this system in the mod m field, we can conclude that there is a unique set of $\{a, b, c\} \in U$ such that

$$\begin{cases} h_{a,b,c}(x_1) = y_1 \\ h_{a,b,c}(x_2) = y_2 \\ h_{a,b,c}(x_3) = y_3 \end{cases}$$

- Since $|\mathcal{H}| = m^3$, and for three fixed distinct number $x_1, x_2, x_3 \in \{1, \ldots, m-1\}$ and three fixed number $y_1, y_2, y_3 \in \{1, \cdots, m-1\}$ there is exactly one $h_{a,b,c} \in \mathcal{H}$ such that $h_{a,b,c}(x_1) = y_1, h_{a,b,c}(x_2) = y_2, h_{a,b,c}(x_3) = y_3$, we know that

$$\text{Prob}_{h_{a,b,c} \in \mathcal{H}}[h_{a,b,c}(x_1) = y_1, h_{a,b,c}(x_2) = y_2, h_{a,b,c}(x_3) = y_3] = \frac{1}{m^3}$$

So by definition, the family $\mathcal{H} = \{h_{a,b,c} | a, b, c \in U\}$ is 3-universal.

Q2

(a) We use an array T with m slots (with indexes being $1, 2, \ldots, m$), where the i-th slot stores a pointer to a node n_i containing three attributes, b, v, w. For each node n_i in a slot of the array,

- When $n_i.b = 0$, i-th slot is "empty". And, $n_i.v$ represents the index of the empty slot previous to it, and $n_i.w$ represents the index of the empty slot after it. Here, "previous" and "after" is just the order of this implicit sequence, can be irrelevant with their order appears in the array.
- When $n_i.b = 1$, this node is not "empty" and stores an element from U. In detail, $n_i.v$ refers to an element in the set U.

  And, $n_i.w$ is set to 0 if there is no element in U stored in T, that is hashed to the same index of the array as element $n_i.v$ and inserted into T after n_i.

  Otherwise, $n_i.w = j$ is a index of array where $T[j]$ stores n_j such that $n_j.v$ is the element in S that has the same hash value with $n_i.v$ and is the one inserted into T after $n_i.v$, among all collided elements.

So, we basically construct implicit linked lists using v and w as pointers, with information stored in the array and its pointers to nodes.

One doubly linked list (also circular, with head and tail connected by their attributes) representing a list of slot that are "empty" (its node stored with $b = 0$), and that v is like the *prev* attribute, and w is like the *next* attribute, are indexes of the empty slot previous to/next to it in the table.

Some singly linked lists (this is where the chaining of collided elements come in), each represent a sequence of nodes with collision to the same slot in T, connected by w attributes (like *next*). That is, for $u_1 \in S$, to find the slot that stores the element u_2 in S collides with u_1 and insert into T after u_1, we can use $u_1.w$ as accessing index to T, i.e., $T[u_1.w]$, and that $T[u_1.w].v$ is the element u_2. The last node u_l of this singly linked list has $u_l.w = 0$, representing no next element to u_l in this implicit linked list.

(For **extra memory**, we use a pointer points to the first node of the doubly linked list of the empty slots so that we can find an empty slot in constant time.

We store a pointer to a node in the table because we want to refer to the b, v, w attribute more clearly, actually to save memory, we can just store a tuple (b, v, w) in each slot of the table)

Formally speaking, our data structure have three important property

1. All "empty" slot are being connected circularly like an doubly linked list, with their v and w attributes, points to indexes of other two slots in T (like *prev* and *next*).
2. An extra memory is used to store a pointer to a node in one of the "empty" slots, it points to nil if there is no empty slot in T.
3. Other than those "empty" slots, the rest are slots with its node stores an element in U, by v attribute. And these nodes are nicely grouped/connected, by their w attribute. Much like several singly linked list, each groups elements with same hash value (i.e., chaining for collision).

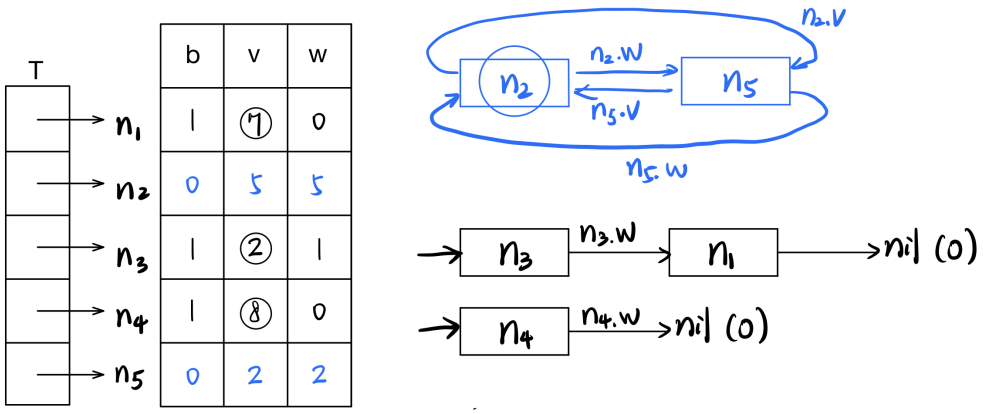
As an example, we show how we represent the implicit linked lists in (b), with some extra linked list graphs beside the array:

$n_1.b = n_3.b = n_4.b = 1$ means the v attributes of these three nodes store the value in S. Since $h(2) = h(7) = 3$, we can only store 2 in slot 3, but $n_3.w = 1$ points to the first slot that stores 7. Only $h(8) = 4$ so $n_4.v = 8$ and $n_4.w = 0$ representing there is no collision at the 4th slot.

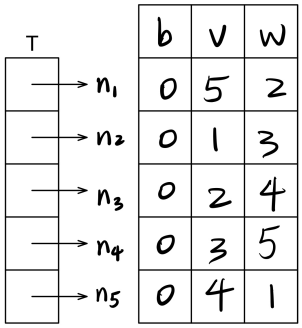
$n_2.b = n_5.b = 0$ means these two slots do not store any value in S. We have $n_2.v = n_2.w = 5$ and $n_5.v = n_5.w = 2$, which means the "previous" and "next" empty slots of the 2nd slot are both the 5th slot and vice versa.

And we use an extra memory to store the pointer to n_2 which is an empty node.

(b)



(c) Denote $T[i]$ to be n_i, i.e., the node stored in slot i of array T, for $i \in \{1, \ldots, m\}$. Notice that as discussed in (a), when S is initially empty (i.e., T stores no element from U), for every node n_i, we should set $n_i.b = 0$. And, the whole table is represented as the doubly empty linked list discussed in (a), and we can just connect them in the order they presents in the array T. So, $n_i.v$ should be the index of empty slot previous to it in the linked list, i.e., $i - 1$ if $1 < i \leq m$, and m if $i = 1$ (wrap around). And $n_i.w$ should be the index of empty slot next to it in the linked list, i.e., $i + 1$ if $1 \leq i < m$, and 1 if $i = m$ (wrap around). Figure below shows a example of T with $m = 5$.



(d)

Firstly, let's define some helper algorithms:

DELETE-EMPTY-SLOT(n) — set the w attribute of the previous empty slot of n (which is $T[n.v]$) to $n.w$, and set the v attribute of the next empty slot of n (which is $T[n.w]$) to its $n.v$ (indeed, a deletion of doubly linked list).

GET-EMPTY-SLOT() — Return the pointer to an empty node n or nil stored in the extra memory and update the pointer to $n.w$. Before return a pointer, we need to call DELETE-EMPTY-SLOT(n) to delete this node from the doubly linked list of empty slots.

After every insertion, we want to preserve the three properties of our data structure described in part (a).

To insert x, we first compute the hash value $h(x)$, and find the corresponding slot in the array.

First, we check if there is any slot still "empty", by checking the extra memory, if it points to NIL, by property 2, we are done since it means the table is full.

Otherwise, there is an empty slot, then we check the b value of node $T[h(x)] = n$ in the slot.

- If $n.b = 0$, we will just set $n.b = 1$ as soon $n.v$ will be set to x and this slot no longer being empty. Before set $n.v$ to be x, we have to delete it from the doubly linked list of "empty" slots first, this can be done through DELETE-EMPTY-SLOT(n) helper algorithm. Then, after all, we can set $n.v$ to be x, and $n.w$ to be 0 (like NIL, indicating no other element in T has same hash value yet, end of this linked list)

- If $n.b = 1$, this slot is occupied ($n.v$ is an element from U) we have two cases to deal with, depending on whether $h(n.v) = h(x)$
  - If $h(n.v) = h(x)$, we find the head of the "singly linked list" of hash value $h(x) = h(n.v)$, so we just insert x into this linked list by imitating standard singly linked list insertion. We call helper algorithm GET-EMPTY-SLOT() to get a pointer to node u in an empty slot, and put x into $u.v$. Then, we set $u.w = n.w$ (set inserted node's next to be header node's next).
  - If $h(n.v) \neq h(x)$, this slot is"illegally" occupied by $n.v$ ($h(n.v) \neq$ index of this slot), so we put current $n.v$ to other place, and put x here (setting $n.v = x$ and $n.w = 0$, representing new linked list with one element). We then call GET-EMPTY-SLOT() to get a pointer to node u in an empty slot, and then we move $n.v$ to the slot that node u resides in, by setting $u.v = x$ and $u.b = 1$ and $w = 0$. And one thing left is connect its "previous" slot to the slot to its "new home". As this is a singly linked list, we have to do it from head. We compute the hash value of current $u.v$ and find the head linked list with this hash value, we then traverse to the end via w attribute, and stops in the slot stores the element previous to $u.v$, say this slot has node u', and we set $u'.w$ to be index of $u.v$'s slot.

This is the whole algorithm, notice that we preserve our data structure properties 1-3. 1 and 2 trivially hold (an empty slot are now occupied by some element). The last case in our algorithm, is for preserving property 3 (creating a new head of a linked list, and moved illegal element elsewhere).

Worst-case expected time analysis:

Consider first few cases, they are all constant running time. For the last case, the running time is proportional to x's chaining, as property 3 suggests, we should traverse through almost all slots with their $n.v$ such $h(n.v)$ equals the moved element. Notice that, by Theorem 11.3 in CLRS, since h is randomly drawn from a universal hash family of functions, the expected length of this chaining/singly linked list with this hash value, would be at most $1 + \alpha$ (the actual theorem gives condition on whether key is in table, but we use the larger upper bound $1 + \alpha$ as it still helps us conclude right running time), and since $\alpha = n/m = |S|/|T| \leq 1$, we have that the expected running time is $O(1 + \alpha) = O(1)$. Since this analysis hold for any x inserted, it is the worst case expected time. So, in all cases, the worst cases expected time is $O(1)$, as needed.

(e) Deletion should be much like singly linked list deletion, with property of our data structure holds.

First of all, to delete an input x, we first compute its hash value, $h(x)$. We find the slot in T, get node $n = T[h(x)]$.

- If $n.b = 1$, the singly linked list of this hash value exist, so x can be in the list. We basically do search on this singly linked list. We set two pointers, u and p, and we traversing through this linked list by w attributes that connects them, using two pointers where $p.w = u$. If in process of looping, $u.v = x$, we find the element we want to delete. We delete it by firstly, set its $p.w = u.w$, and then remove $u.v$ from the singly linked list by setting $u.b = 0$, and inserting it into the doubly linked list of empty slot via pointer in the extra memory (to preserve property 1), since this slot is now empty. The loop stops when $u.w = 0$ (we reach the end of the list).
- If $n.b = 0$, the singly linked list of this hash value do not exist, property suggests that there is no element with this hash value in T, we can stop here.

Worst-case expected time analysis: By Theorem 11.3 from CLRS, since h is randomly chosen from a universal family of hash functions, for any input x, the expected length of the singly linked list with hash value $h(x)$ is at most $1 + \alpha$. Since our algorithm has running time at most proportional to the chaining length $1 + \alpha$, similar to d) we can conclude that the expected time is $O(1+) + O(1 + |S|/|T|) = O(1)$. Notice that this analysis hold for any input x, so the worst-case expected time is $O(1)$.

(f) Searching for an element $x \in U$ in our array T, is almost the same as searching in a singly linked list. (We have briefly introduced the searching method in part f)

Firstly, compute hash value of x and find the slot $T[h(x)]$, this is the head of the linked list we search for (as property 3 suggests, if x in the table, it must be in the list starting by this slot). Similar to algorithm described above, but we just need one pointer u during traversing this time. We loop until either $u.v = x$ (we successfully found it) or $u.w = 0$ (we have reached the end of this linked list, if this $u.v \neq x$, x should not be in T so this is an unsuccessful search).

Worst-case expected time analysis: Similar to deletion, by Theorem 11.3 from CLRS, since h is randomly chosen from a universal family of hash functions, for any input x, the expected length of the singly linked list with hash value $h(x)$ is at most $1 + \alpha$. In either successful or unsuccessful search, the expected running time of our algorithm is proportional to this expected chaining length in $O(1 + \alpha)$. And since this analysis hold for all $x \in U$, the worst-case expected time is $O(1)$.