Hashing

1. Introduction:

- Many applications require a dynamic set that supports only the dictionary operations Insert, Search and Delete.
- A hash table is an effective data structure for implementing dictionaries.
- Although searching for an item in a hash table can take $\Theta(n)$ time in the worst case, in practice, hashing performs extremely well. Under reasonable assumptions, the avg time to search for an element in a hash table is $O(1)$.

2. Direct Addressing:

- Direct addressing is a simple technique that works well if the number of keys is reasonably small.

- Suppose that an application needs a dynamic set in which each element has a key drawn from the universe $U = \{0, 1, \ldots, m-1\}$ where m is not too large. Furthermore, assume that no two elements have the same key. To represent the dynamic set, we can use an array or a **direct-address table**, denoted by $T[0, \ldots, m-1]$. Each position or spot corresponds to a key in U. If the set contains no elements with key k, then $T[k] =$ NULL.

- The dictionary operations are trivial to implement.

```
insert(T, x):
    T[x.key] = x

search(T, key):
    return T[key]

delete(T, x):
    T[x.key] = NULL
```

Each operation takes $\Theta(1)$ time in the worst case.

- Suppose that we know the key-values are integers from 1 to k. A simple and fast way to rep a dictionary is to allocate an array of size k and store an element with key i in the i^{th} cell of the array.

- The problem with direct addressing is that it only works well if the number of keys are small. If the number of keys is too big, then the array will be huge, which is space efficient.

- E.g. 1 Suppose we are reading a text file and want to store the freq of each letter of the text file. Why is this a good application of direct addressing?

  Soln:

  Since there are only 256 ASCII chars, we can make an array with a length of 256. Furthermore, we can let the i^{th} cell hold the number of occurrences of the i^{th} ASCII character.

- E.g. 2 Suppose we are reading a data file of 32-bit ints and we want to track the freq of each number. Why is this a bad application of direct addressing?

Soln:

The array would have a size of 2^{32}.

We can use a hash table to solve this problem.

3. Hash Tables:

- Suppose that the key-values of our elements come from a universe or set U.

- We can allocate a table or array of size m, where $m < |U|$.

- Then, we can use a **hash function** $h: U \rightarrow \{0, \ldots, m-1\}$ to decide where to store the element with key-value x. I.e. x gets stored in position h(x) of the hash table.

- E.g. Consider E.g. 2. Given the set of keys is the set of all integer values from 0 to $2^{32}-1$, what is a possible hash function if $m = 2^{20}$?

  Soln:

  We can use either key mod m or key ÷ m.

  The idea is that when we want to access key k, we look in T[f(k)] instead of T[k].

- If $m < |U|$, then there must be some keys, k_1 and k_2, in U s.t. $k_1 \neq k_2$ and yet $h(k_1) = h(k_2)$. This is called a **collision**.

Collision Resolution:

- E.g. You have an address book and the N section fills up. Where do you put the next N entry?

  Soln:

  1. You flip to the next page and have an overflow page. (Open Addressing)

  2. Write a note explaining where to find the rest of the N entries. (Closed Addressing)

- A **collision** occurs when a hash function maps 2 keys to the same index.

- We will look at 2 general solutions:

  1. **Closed Addressing**: Give explicit directions of the next location.

  2. **Opening Addressing**: Give a general rule of where to look next.

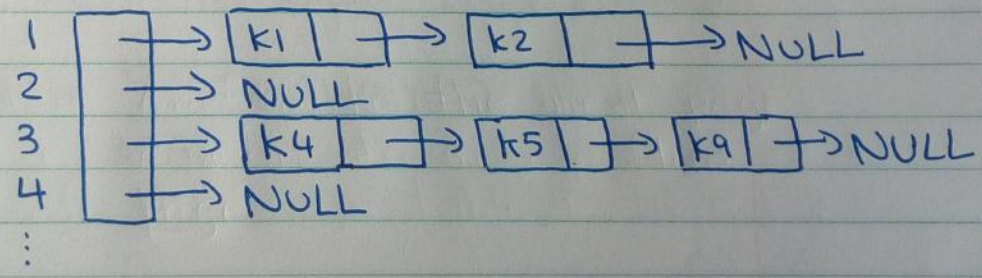
5. Closed Addressing:

- We can resolve a collision using **chaining**. By chaining, I mean storing a linked list at each entry in the hash table.

- E.g. Suppose the following:

  1. $h(k_1) = h(k_2) = 1$
  2. $h(k_4) = h(k_5) = h(k_9) = 3$

  Then:



- Assume we can compute the hash function, h, in constant time. Then,
  - Insert takes $\Theta(1)$ time.
  - Delete takes $\Theta(1)$ time.
  - Search is a bit more complicated.
- If $m(n-1) < |U|$, then any given hash function will put at least n key-values into some entry of the hash table.
- The worst case scenario occurs when 1 entry has all the elements. Then, search will take $\Theta(n)$ time.
- For the avg case, the sample space is the set of elements that have key-values from U.
- For any prob dist on U, we assume that our hash function, h, obeys a property called **simple uniform hashing**. This means that if $A_i = \{k \in U \mid h(k) = i\}$, A_i are all those members k of U s.t. the hash function computes the same index i, then

$$Pr(A_i) = \sum_{k \in A_i} Pr(h(k) = i)$$
$$= \frac{1}{m}$$

I.e. The simple uniform hashing property means that any given key is equally likely to hash into any of the m slots.

- If the table has n elements, there is expected to be n/m elements in any one entry. This ratio, n/m, is called the **load factor**, denoted by α.

- Note: This assumption may or may not be accurate depending on U, h and the prob dist.

- To calculate the avg-case run time:
  - Let T be a ran var which counts the number of elements checked when searching for key k.
  - Let L_i be the length of the list at entry i in the hash table.
  - Then, the avg-case running time is

$$E(T) = \sum_{k \in U} Pr(\text{Getting key } k) \cdot (\text{num of elements checked during search for } k)$$

$$= \sum_{k \in U} Pr(k) \cdot T(k)$$

$$= \sum_{i=0}^{m-1} \sum_{k \in A_i} Pr(k) \cdot T(k)$$

Splits U into disjoint sets A_i for all i.

$$\leq \sum_{i=0}^{m-1} Pr(A_i) L_i$$

$$= \frac{1}{m} \sum_{i=0}^{m-1} L_i$$

$$= \frac{n}{m}$$

$$= a$$

- The above calculation is really the expected time for k most likely not in L_j, $\forall j$.

- If $k \in L_j$, for some j, then we want to consider k chosen uniformly at random from the elements in T.

$$Pr[h(k) = i] = \frac{L_i}{n}$$

- Furthermore, the probability that k is the j^{th} element in bucket i conditional to the fact that $h(k) = i$ is $\frac{1}{L_i}$.

- $E[T] = \sum_{k \in T} Pr(k) \cdot T(k)$

$$= \sum_{i=1}^{m-1} \left(Pr(h(k) = i) \cdot \sum_{j=1}^{L_i} (j \cdot Pr(k \in j^{th} \text{ slot in list } i)) \right)$$

$$= \sum_{i=1}^{m-1} \left(\frac{L_i}{n} \sum_{j=1}^{L_i} \frac{j}{L_i} \right)$$

$$= \frac{1}{n} \sum_{i=1}^{m-1} \sum_{j=1}^{L_i} j$$

$$= \frac{1}{n} \sum_{j=0}^{m-1} (L_i)\left(\frac{L_i+1}{2} \right)$$

$$= \frac{1}{2n} \sum_{i=0}^{m-1} (L_i)^2 + \frac{1}{2n} \sum_{i=0}^{m-1} L_i$$

$$= \frac{1}{2n} \sum_{i=0}^{m-1} (L_i)^2 + \frac{1}{2}$$

$$= \frac{1}{2n} \sum_{i=0}^{m-1} \left(\frac{n}{m}\right)^2 + \frac{1}{2}$$

$$= \frac{n^2}{2nm} + \frac{1}{2}$$

$$= \frac{\alpha+1}{2}$$

$\therefore$ The avg-case running time of Search under simple uniform hashing with chaining is $O(\alpha)$.

Depending on the application, α can sometimes be a constant if m is big enough. In this case, search takes $O(1)$ time on avg.

6. Open Addressing:

- Each entry in the hash table stores a fixed number of c elements. For simplicity, assume that c=1. We only use open addressing if $cn < m$.
- If we get a collision when we insert a new element:
  1. We find a new location to store the new element.
  2. We need to know where we put it. To do this, we search for a well-defined sequence of other locations in the hash table until we find one that's not full.

     This sequence is called a **probe sequence.**

7. Probe Sequences:

- There are 3 methods for generating a probe sequence:
  1. **Linear Probing:** Try $A[(h(k) + i) \bmod m]$ where $i = 0, 1, \ldots$
  2. **Quadratic Probing:** Try $A[(h(k) + c_1 i + c_2 i^2) \bmod m]$

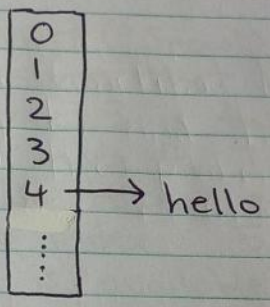
3. Double Hashing: Try $A[(h(k) + ih'(k)) \bmod m]$ where h' is another hash function.

- Linear Probing:
  - The easiest open addressing strategy.
  - For a hash table of size m, key k, and hash function h(k), the probe seq is calculated as $s_i = (h(k) + i) \bmod m$ for $i = 0, 1, \ldots$

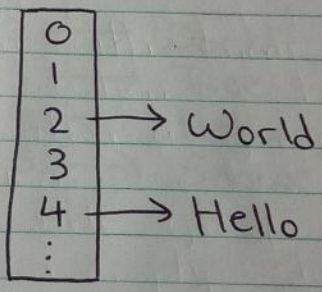
    Note: The value of s_0, the home location for the item, is h(k).
  - The problem with linear probing is clustering. When we hash to something within a group of filled locations, we have to probe the whole group until we reach an empty slot. This increases the size of the cluster. Overall, this results in 2 keys that didn't necessarily share the same "home" location ending up with almost identical probe sequences.

- E.g. Suppose we start off with an empty array. Consider this sequence of inserts.

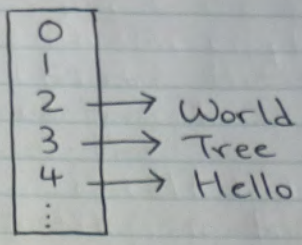
1. insert("hello")
   Assume that h("hello") = 4



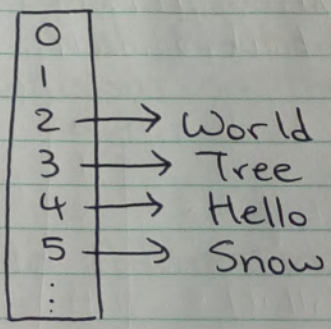
2. insert("world")
   Assume that h("world") = 2



3. insert("tree")
   Assume that h("tree") = 2
   However, world is already in index 2. We have to go to index 3 to see if it's empty. Since index 3 is empty, we put tree there.



4. Insert("Snow")
Assume that h("snow") = 3
Since 3 is taken, we go to index 4. Index 4 is also taken, so we go to index 5. Index 5 is available, so we insert snow there.



- Quadratic Probe:
  - The probe seq is calculated as $S_i = (h(k) + C_1 i + C_2 i^2) \bmod m$ for $i = 0, 1, 2, \ldots$
  - Quadratic probing faces the same problem as linear probing; Clusters.

- Double Hashing:
  - Use a second, different hash function, $h_2(k)$, to calculate the step size.
  - The probe sequence is $s_i = (h(k) + i \cdot h_2(k)) \bmod m$ for $i = 0, 1, 2, \ldots$
  - Note: $h_2(k)$ shouldn't be 0 for any value of k.
  - We want to choose a h_2 s.t. even if $h(k_1) = h(k_2)$, it won't be the case that $h_2(k_1) = h_2(k_2)$. This way, the two hash functions don't cause collisions on the same pairs of keys.

8. Analysis of Open Addressing:

- In open addressing insert and search takes $\Theta(n)$ in the worst case.
- To simplify the average case analysis, we assume the following:
  1. The hash table has m locations.
  2. The h.t. contains n elements and we want to insert a new key, k.
  3. We consider a random probe seq for k. I.e. The probe seq is equally likely to be any permutation of $(0, 1, \ldots m-1)$.

- Let T denote the number of probes performed in Insert. Then, the avg-case time for insert is the expected-time (E(T)).

$$E(T) = \sum_{i=0}^{m-1} i \Pr(T=i)$$

$$\Pr(T=i) = \Pr(T \geq i) - \Pr(T \geq i+1)$$

Let A_i denote the event that every location up until the i^{th} probe is occupied. Then, $T \geq i$ iff $A_1, A_2, \ldots A_{i-1}$ all occur. This means that

$$\begin{aligned}\Pr(T \geq i) &= \Pr(A_1 \cap A_2 \ldots \cap A_{i-1}) \\ &= \Pr(A_1)\Pr(A_2|A_1)\Pr(A_3|A_1 \cap A_2)\ldots \\ &\quad \Pr(A_{i-1}|A_1 \cap \ldots \cap A_{i-2})\end{aligned}$$

For $j \geq 1$, $\Pr(A_j|A_1 \cap \ldots \cap A_{j-1}) = \frac{n-(j-1)}{m-(j-1)}$

This is because we need the number of elements that we have not seen so far over the number of slots we have not seen so far.

$$\begin{aligned}\Pr(T \geq i) &= \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)\cdots\left(\frac{n-(i-2)}{m-(i-2)}\right) \\ &\leq (n/m)^{i-1} \\ &\leq a^{i-1}\end{aligned}$$

9 10 11

$$- E(T) = \sum_{i=0}^{m-1} i \Pr(T=i)$$

$$\leq \sum_{i=1}^{\infty} i \Pr(T=i)$$

$$\leq \sum_{i=1}^{\infty} i(\Pr(T \geq i) - \Pr(T \geq i+1))$$

$$\leq \sum_{i=1}^{\infty} i \Pr(T \geq i) - \sum_{i=1}^{\infty} i \Pr(T \geq i+1)$$

$$\leq \sum_{i=1}^{\infty} i \Pr(T \geq i) - \left(\sum_{i=1} (i+1) \Pr(T \geq i+1) - \right.$$
$$\left. \Pr(T \geq i+1) \right)$$

$$\leq \sum_{i=1}^{\infty} i \Pr(T \geq i) - \sum_{i=2} i \Pr(T \geq i) + \sum_{i=2} \Pr(T \geq i)$$

$$\leq \Pr(T \geq 1) + \sum_{i=2} i \Pr(T \geq i) - \sum_{i=2} i \Pr(T \geq i)$$

$$+ \sum_{i=2} \Pr(T \geq i)$$

$$\leq \Pr(T \geq 1) + \sum_{i=2} \Pr(T \geq i)$$

$$\leq \sum_{i=1}^{\infty} \Pr(T \geq 1)$$

$$\leq \sum_{i=1}^{\infty} a^{i-1}$$

$$\leq \sum_{i=0}^{\infty} a^i \quad \leq \frac{1}{1-a}$$

Since $n<m$, $a<1$. This means that the bigger the load factor, the longer it takes to insert something.

- Under open addressing, there are 2 approaches for remove:
  1. Find an existing key to fill the hole. This is tricky for probing and impossible for double hashing.
  2. We mark the cell as deactivated, not as free.
- Each cell has 3 possibilities:
  1. Free: Can insert and stop searching here.
  2. Deactivated: Can insert here but cannot stop searching here.
  3. Stores a key.
- Remove is problematic under open addressing as it accumlates junk and slows down all operations.

9. Hash Functions:

- There are 4 hash functions that we will look at. They are:
  1. Division Method
  2. Multiplication Method
  3. Polynomial Hash
  4. Fowler-Noll-Vo (FNV-1)

- Division Method:
  - Assume that each key is an integer.
  - $h(k) = k \bmod m$. $h(k)$ will be between 0 and $m-1$.
  - Although this is simple, it is susceptible to regular patterns in keys. This means that there will be a lot of collisions. I.e. The division method is sensitive to the value of m.
  - To overcome this issue, we pick m to be a prime number.

- Multiplication Method:
  - $h(k) = \lfloor m \cdot \text{fraction}(k \cdot A) \rfloor$
  - In theory, we want to pick a real constant A s.t. $0 < A < 1$.

- However in practice, we assume that each key is a w-bit natural number. We define A by picking a w-bit constant S, s.t. $0 < S < 2^w$, and letting $A = \frac{S}{2^w}$.

Furthermore, we let $m = 2^p$, for some p s.t. $0 \leq p < w$. Then:

$$h(k) = \lfloor m \cdot \text{fraction}(k \cdot A) \rfloor$$

$$= \lfloor 2^p \cdot \text{fraction}(k \cdot S / 2^w) \rfloor$$

$$= \left\lfloor 2^p \left(\frac{(k \cdot S) \bmod 2^w}{2^w} \right) \right\rfloor$$

$$= \left\lfloor \frac{(k \cdot S) \bmod 2^w}{2^{w-p}} \right\rfloor$$

$(k \cdot S) \bmod 2^w$ returns the lower w bits of $(k \cdot S)$. Then, dividing by 2^{w-p} returns the upper p bits of the lower w bits of $(k \cdot S)$.

- We often want A to be irrational, so we often use the golden ratio for A and work backwards to define S.

- Polynomial Hash:
  - Used to hash string typed keys.
  - A string is a seq of chars encoded using ASCII code.
    E.g. H E L L O
    72 69 76 76 79
  - Given a string $s = x_0, \ldots, x_{k-1}$,
    $h(s) = (x_0 \cdot a^{k-1} + x_1 \cdot a^{k-2} + \ldots + x_{k-2} \cdot a + x_{k-1}) \bmod m$.
  - a must be a non-zero constant and $a \neq 1$. I.e. a cannot be 0 or 1. Some good values of a are 33, 37, 39 and 41.
  - Pseudo Code:

```
c = 0
for i in 0 ... k-1:
    c = c · a + x_i
c = c mod m
```

- FNV-1:
  - FNV-1 is a family of hash functions, one for each word size.
    I.e. FNV comes in 32, 64, 128, 256, 512 and 1024 bit flavours.

- An adv of FNV-1 is that it's very easy to implement. You start with an inital hash value called the FNV offset basis. For each byte in the input, you multiply hash by the FNV prime. Finally, you XOR it with the byte from the input.
- They work by chopping the key into 8-bit words.
- Pseudo Code for 32-bit FNV-1:

```
hash = 2166136261    This is the FNV offset basis.

for i in 0... k-1:
    For each byte of data.
    hash = hash * 0x01000193
    Here, we are multiplying hash by the FNV Prime, 0x01000193.
    hash = hash XOR byte;
    Here, we are XORing hash with the byte from the input.
```

- FNV-1a:
  - Like FNV-1 but you xor the hash value with the byte from the input before you multiply hash with the FNV prime.
  - Pseudo Code for 32-bit FNV-1a:

```
hash = 2166136261
for i in 0.. k-1:
    hash = hash XOR byte_i
    hash = hash · 0x01000193
```

  - FNV-1a is recommended over FNV-1 for being more random and uniform.

- The problems with hashing are that when the set of keys is unknown, we can no longer assume a uniform distributions and regular patterns can be found, making any deterministic hashing scheme vulnerable to malicious slowdowns.

- A solution to the above problem is to create a family of hash functions and to select one at random. This is called universal hashing.

10. Universal Hashing

- Definition:
  - A family H of hash functions is universal iff:

    For any 2 keys j and k s.t. $j \neq k$ and a table of size m,

    1. At most $\frac{|H|}{m}$ functions satisfy $h(j) = h(k)$.
       I.e. we randomly pick h from H with uniform probability.

    2. $Pr(h(j) = h(k)) = \frac{1}{m}$

       This is because
       $Pr(h(j) = h(k))$
       $= \frac{\#\text{ of functions that map } j \text{ and } k \text{ to the same place}}{\text{Total \# of functions}}$
       $= \frac{\frac{|H|}{m}}{|H|}$
       $= \frac{1}{m}$

  - Given a set of keys, we randomly select a hash function h() from H and use this function for every key in our set.

- Expected Number of Collisions:
  - If there are n keys hashed to a table of size m, there will be $O(\frac{n}{m})$ collisions.
  - Proof:

Let S be a set of n keys.
Let j be a key not in S.
Randomly pick h from H.
Let the r.v. C be the num of collisions.
For each $k \in S$, let the indicator r.v. X_k be 1 when j collides with k and 0 otherwise.

$$E(C) = E\left(\sum_{k \in S} X_k\right)$$

$$= \sum_{k \in S-j} E(X_k)$$

$$= \sum_{k \in S-j} Pr(h(j) = h(k))$$

$$\leq \sum_{k \in S-j} 1/m$$

$$= \frac{n-1}{m}$$

$$< \frac{n}{m}$$

$$\in O(n/m)$$

- A universal family:
  - Find a prime p large enough s.t. $m \leq p$ and every key k (Assume it's an integer) satisfies $0 \leq k < p$.
  - We define the following:
    1. $f_{a,b}(k) = (ak + p) \bmod m$
    2. $h_{a,b}(k) = f_{a,b}(k) \bmod m$
       $= ((ak + p) \bmod m) \bmod m$
  - We define a universal family of hash functions, H, to be:
    $H = \{h_{a,b} : (0 < a < p) \wedge (0 \leq b < p)\}$.
  - If we randomly pick a from 1 to $p-1$ and b from 0 to $p-1$, there are $(p-1) \cdot p$ choices.
  - To prove that H is a universal family, we need to show that for keys k and j s.t. $k \neq j$, $\Pr(h(j) = h(k)) \leq 1/m$.
    I.e. $\Pr(h(j) = h(k))$
    $\leq \frac{\text{number of } (a,b) \text{ collisions}}{\text{number of } (a,b) \text{ pairs}}$
    $\leq \frac{1}{m}$

- First, we have to prove that $f_{a,b}$ has no collisions. I.e. We want to show that for keys j, k s.t. $j \neq k$, for every (r,s) with $0 \leq r < p$, $0 \leq s < p$ and $r \neq s$, there exists a unique (a,b) s.t. $f_{a,b}(j) = (aj+b) \bmod p = r$ and $f_{a,b}(k) = (ak+b) \bmod p = s$.

This means that there's a 1-1 correspondence between (a,b) and (r,s). This means that to count the number of a and b pairs that cause a collision, we can count the number of r and s pairs s.t. $r \cong s \bmod m$.

Claim: Let j and k be diff keys. Then, $f_{a,b}(j) \neq f_{a,b}(k)$.

Proof:
- We can prove this using a proof by contradiction.
- Without loss of generality, assume that $k < j$.
- Suppose $f_{a,b}(j) = f_{a,b}(k)$.
- Then, $aj+b-(ak+b)$ is a multiple of p.
- This means that $a(j-k)$ is a multiple of p. Then, a or $j-k$ is a multiple of p, because p is prime.

- But $0 < a < p$ and $0 < j-k < p$. Neither can be a multiple of p. This is a contradiction.

- Next, I will prove that there is a 1-1 correspondence between (a,b) and (r,s).

Claim: Given keys j, k with $j \neq k$, for every (r,s) with $0 \leq r < p$, $0 \leq s < p$, $r \neq s$, there exists an unique (a,b) s.t. $f_{a,b}(j) = (aj+b) \bmod p = r$ and $f_{a,b}(k) = (ak+b) \bmod p = s$.
I.e. There is a 1-1 correspondence between (a,b) and (r,s).

Proof:
- We can prove this by a proof by contradiction.
- Suppose that there are (a_1, b_1) and (a_2, b_2) s.t. $a_1 \neq a_2$, $b_1 \neq b_2$, $f_{a_1,b_1}(j) = f_{a_2,b_2}(j)$ and $f_{a_1,b_1}(k) = f_{a_2,b_2}(k)$.
- This means that $(a_1 j + b_1) \equiv (a_2 j + b_2) \pmod p$ and $(a_1 k + b_1) \equiv (a_2 k + b_2) \bmod p$.
- This means that

$$\frac{a_2 j + b_2 - (a_1 j + b_1)}{p} = X, \; X \in Z$$

and

$$\frac{a_2 k + b_2 - (a_1 k + b_1)}{p} = Y, \; Y \in Z$$

- This means that
  $a_2j + b_2 - a_1j - b_1 = xp$
  and
  $a_2k + b_2 - a_1k - b_1 = py$
- This means that
  $(a_2 - a_1)j + (b_2 - b_1) = xp$
  and
  $(a_2 - a_1)k + (b_2 - b_1) = py$
- However, we know that $0 < a_i < p$, $0 \leq b_i < p$. This means that the only way the above eqns are correct is if $a_1 = a_2$ and $b_1 = b_2$. This is a contradiction.

- Since it's proven that $f_{a,b}$ does not cause any collisions, that means $h_{a,b}$ must cause the collisions.

We want to prove that given keys j and k s.t. $j \neq k$ and random a, b, the number of collisions is

$$|\{(a,b) : h_{a,b}(j) = h_{a,b}(k)\}|$$
$$\leq \frac{p(p-1)}{m}$$
$$= \frac{|H|}{m}$$

Proof:

$$
\begin{aligned}
&|\{(a,b): h_{a,b}(j) = h_{a,b}(k)\}| \\
&= |\{(a,b): f_{a,b}(j) \bmod m = \\
&\qquad f_{a,b}(k) \bmod m\}| \\
&= |\{(r,s): r \neq s \wedge r \bmod m = s \bmod m\}| \\
&= \sum_{r=0}^{p-1} |\{s: r \neq s \wedge r \bmod m = s \bmod m\}| \\
&= \sum_{r=0}^{p-1} |\{s: r \bmod m = s \bmod m\}| - 1 \\
&= \sum_{r=0}^{p-1} \lceil p/m \rceil - 1
\end{aligned}
$$

$$
\lceil p/m \rceil \in \{p/m, (p+1)/m, \ldots (p+m-1)/m\}
$$

$$
\begin{aligned}
&= \sum_{r=0}^{p-1} (p+m-1)/m - m/m \\
&= \sum_{r=0}^{p-1} (p-1)/m \\
&= \frac{p(p-1)}{m} \\
&= \frac{|H|}{m}
\end{aligned}
$$