COMP 761: Lecture 30 – Hashing

David Rolnick

November 13, 2020

Problem

Suppose we have integers $k_1 \neq k_2$, a prime number p, and integers a, b with a not divisible by p. Define the remainders mod p:

$$\begin{aligned} r_1 &= (ak_1 + b) \mod p \\ r_2 &= (ak_2 + b) \mod p. \end{aligned}$$

Prove that $r_1 \neq r_2$, and that it is possible to solve for a and b modulo p given p, r_1, r_2, k_1, k_2.

(Please don't post your ideas in the chat just yet, we'll discuss the problem soon in class.)

Course Announcements

Course Announcements

- Problem 3 – you can assume that you know the max flow already. (If you didn't know the max flow, the algorithm might take longer than $|E|$ steps.)

Arrays

Arrays

- An *array* is a way of storing information in memory.

Arrays

- An *array* is a way of storing information in memory.
- Creating an array with a certain length n specifies a block of n spots in memory where things can be stored.

Arrays

- An *array* is a way of storing information in memory.
- Creating an array with a certain length n specifies a block of n spots in memory where things can be stored.
- An array has *random access* – that is, you can easily look at spot k out of n for any k.

Arrays

- An *array* is a way of storing information in memory.
- Creating an array with a certain length n specifies a block of n spots in memory where things can be stored.
- An array has *random access* – that is, you can easily look at spot k out of n for any k.
- Inserting at the end of an array is easy – until you exceed the allocated space.

Arrays

- An *array* is a way of storing information in memory.
- Creating an array with a certain length n specifies a block of n spots in memory where things can be stored.
- An array has *random access* – that is, you can easily look at spot k out of n for any k.
- Inserting at the end of an array is easy – until you exceed the allocated space.
- Inserting at the beginning/middle of an array is a pain, requires shifting everything afterwards (time complexity can be $O(n)$).

Arrays

- An *array* is a way of storing information in memory.
- Creating an array with a certain length n specifies a block of n spots in memory where things can be stored.
- An array has *random access* – that is, you can easily look at spot k out of n for any k.
- Inserting at the end of an array is easy – until you exceed the allocated space.
- Inserting at the beginning/middle of an array is a pain, requires shifting everything afterwards (time complexity can be $O(n)$).
- Similarly, deletion is a pain.

Linked lists

Linked lists

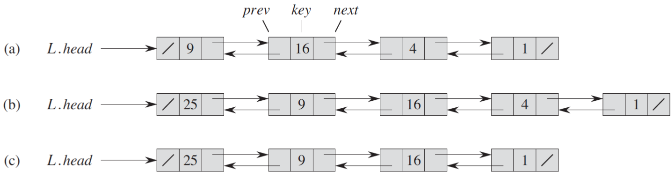
- A *linked list* makes insertion and deletion easier, but loses random access.

Linked lists

- A *linked list* makes insertion and deletion easier, but loses random access.
- Each element includes a pointer to the next element of the linked list.

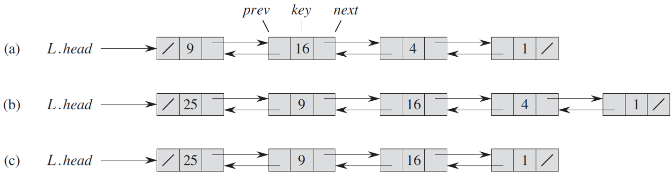
Linked lists

- A *linked list* makes insertion and deletion easier, but loses random access.
- Each element includes a pointer to the next element of the linked list.
- In a *doubly linked list* it also includes a pointer to the previous element.



Linked lists

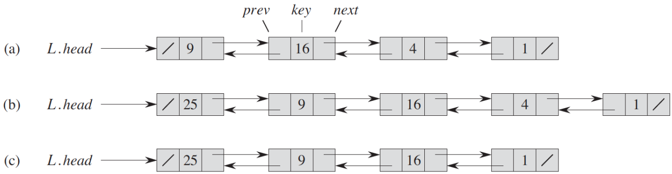
- A *linked list* makes insertion and deletion easier, but loses random access.
- Each element includes a pointer to the next element of the linked list.
- In a *doubly linked list* it also includes a pointer to the previous element.



- So it is possible from any element to access the next one and previous one (*sequential access*), but not to jump immediately to any element.

Linked lists

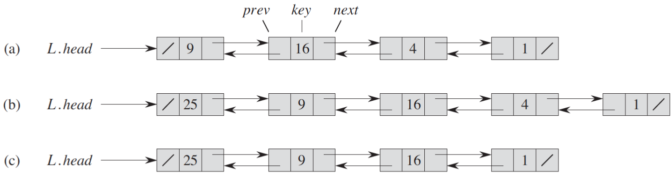
- A *linked list* makes insertion and deletion easier, but loses random access.
- Each element includes a pointer to the next element of the linked list.
- In a *doubly linked list* it also includes a pointer to the previous element.



- So it is possible from any element to access the next one and previous one (*sequential access*), but not to jump immediately to any element.
- Getting the *m*th element takes $O(m)$ time.

Linked lists

- A *linked list* makes insertion and deletion easier, but loses random access.
- Each element includes a pointer to the next element of the linked list.
- In a *doubly linked list* it also includes a pointer to the previous element.



- So it is possible from any element to access the next one and previous one (*sequential access*), but not to jump immediately to any element.
- Getting the mth element takes $O(m)$ time.
- But insertion and deletion are $O(1)$, just insert and rearrange the pointers from the next and previous elements.

Direct addressing

Direct addressing

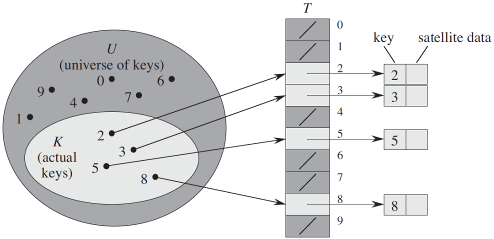
- Sometimes, we will want to store information associated to a given key.

Direct addressing

- Sometimes, we will want to store information associated to a given key.
- For example, the key might be a person's username, and the information might be their records.

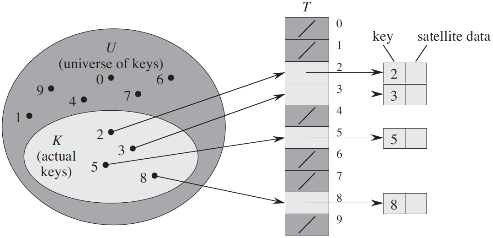
Direct addressing

- Sometimes, we will want to store information associated to a given key.
- For example, the key might be a person's username, and the information might be their records.
- One way to do this is *direct addressing*, using the key as the exact pointer to the records.



Direct addressing

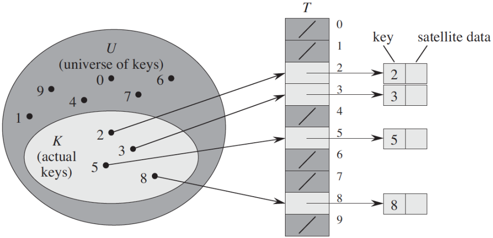
- Sometimes, we will want to store information associated to a given key.
- For example, the key might be a person's username, and the information might be their records.
- One way to do this is *direct addressing*, using the key as the exact pointer to the records.



- What's a potential problem with this?

Direct addressing

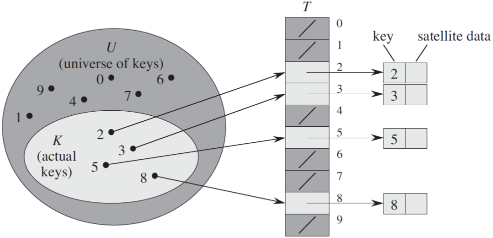
- Sometimes, we will want to store information associated to a given key.
- For example, the key might be a person's username, and the information might be their records.
- One way to do this is *direct addressing*, using the key as the exact pointer to the records.



- What's a potential problem with this?
- There might be a lot of possible usernames.

Direct addressing

- Sometimes, we will want to store information associated to a given key.
- For example, the key might be a person's username, and the information might be their records.
- One way to do this is *direct addressing*, using the key as the exact pointer to the records.



- What's a potential problem with this?
- There might be a lot of possible usernames.
- Would need a huge amount of memory and most of the slots would be unused.

Hashing

Hashing

- In *hashing*, there is a function h that takes possible keys k (e.g. usernames) as input.

Hashing

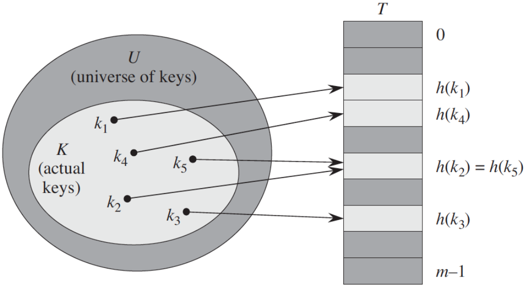
- In *hashing*, there is a function h that takes possible keys k (e.g. usernames) as input.
- The output $h(k)$ is the address for the info stored for k.

Hashing

- In *hashing*, there is a function h that takes possible keys k (e.g. usernames) as input.
- The output $h(k)$ is the address for the info stored for k.
- Typically, number of possibilities for $h(k)$ is much smaller than number of possibilities for k.

Hashing

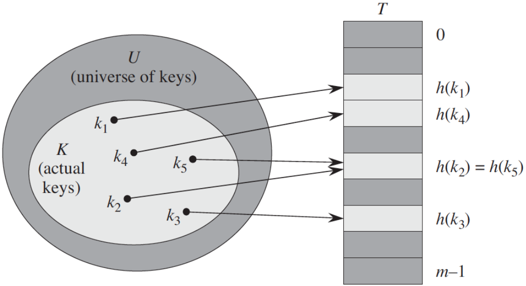
- In *hashing*, there is a function h that takes possible keys k (e.g. usernames) as input.
- The output $h(k)$ is the address for the info stored for k.
- Typically, number of possibilities for $h(k)$ is much smaller than number of possibilities for k.
- This is called a *hash table*.



- What's a potential problem with this?

Hashing

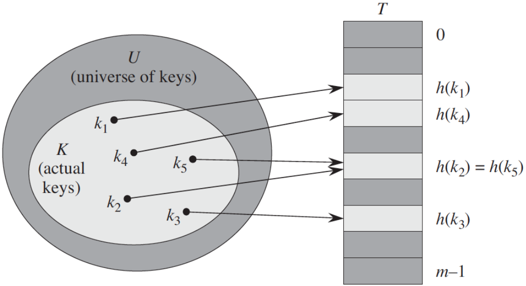
- In *hashing*, there is a function h that takes possible keys k (e.g. usernames) as input.
- The output $h(k)$ is the address for the info stored for k.
- Typically, number of possibilities for $h(k)$ is much smaller than number of possibilities for k.
- This is called a *hash table*.



- What's a potential problem with this?
- We could have k_1 and k_2 where $h(k_1) = h(k_2)$ – which gets stored there?

Hashing

- In *hashing*, there is a function h that takes possible keys k (e.g. usernames) as input.
- The output $h(k)$ is the address for the info stored for k.
- Typically, number of possibilities for $h(k)$ is much smaller than number of possibilities for k.
- This is called a *hash table*.



- What's a potential problem with this?
- We could have k_1 and k_2 where $h(k_1) = h(k_2)$ – which gets stored there?
- This is called a *collision*.

Collisions

Collisions

- One way to try to fix this: Store a linked list at each spot.

Collisions

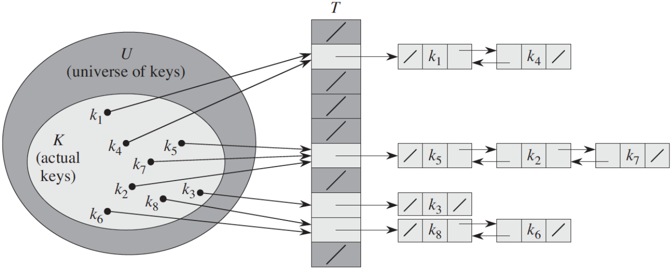
- One way to try to fix this: Store a linked list at each spot.
- First thing inserted there (key k_1) initializes the list.

Collisions

- One way to try to fix this: Store a linked list at each spot.
- First thing inserted there (key k_1) initializes the list.
- If $h(k_2) = h(k_1)$, then add a new entry for k_2 to the start of the list.

Collisions

- One way to try to fix this: Store a linked list at each spot.
- First thing inserted there (key k_1) initializes the list.
- If $h(k_2) = h(k_1)$, then add a new entry for k_2 to the start of the list.
- This is called *chaining* within the hash table.



- Hopefully none of the lists gets very long, since finding something within a linked list is slow.

Analysis

Analysis

- Let's work with the assumption of *simple uniform hashing*, that is, the keys are drawn from a distribution that makes them equally likely to go in any slot, independent of the other keys.

Analysis

- Let's work with the assumption of *simple uniform hashing*, that is, the keys are drawn from a distribution that makes them equally likely to go in any slot, independent of the other keys.
- We therefore have equal probabilities $\Pr(h(k) = x)$ for all x.

Analysis

- Let's work with the assumption of *simple uniform hashing*, that is, the keys are drawn from a distribution that makes them equally likely to go in any slot, independent of the other keys.
- We therefore have equal probabilities $\Pr(h(k) = x)$ for all x.
- If we are storing n keys, with m slots, what is the expected number of keys stored in a single slot?

Analysis

- Let's work with the assumption of *simple uniform hashing*, that is, the keys are drawn from a distribution that makes them equally likely to go in any slot, independent of the other keys.
- We therefore have equal probabilities $\Pr(h(k) = x)$ for all x.
- If we are storing n keys, with m slots, what is the expected number of keys stored in a single slot?
- The expected number in each of the m slots is n/m.

Analysis

- Let's work with the assumption of *simple uniform hashing*, that is, the keys are drawn from a distribution that makes them equally likely to go in any slot, independent of the other keys.
- We therefore have equal probabilities $\Pr(h(k) = x)$ for all x.
- If we are storing n keys, with m slots, what is the expected number of keys stored in a single slot?
- The expected number in each of the m slots is n/m.
- We write $\alpha = n/m$, and call it the *load factor*.

Analysis

- Let's work with the assumption of *simple uniform hashing*, that is, the keys are drawn from a distribution that makes them equally likely to go in any slot, independent of the other keys.
- We therefore have equal probabilities $\Pr(h(k) = x)$ for all x.
- If we are storing n keys, with m slots, what is the expected number of keys stored in a single slot?
- The expected number in each of the m slots is n/m.
- We write $\alpha = n/m$, and call it the *load factor*.
- If we are given a new key k, what is the expected time it takes to search to see if k is in the hash table?

Analysis

- Let's work with the assumption of *simple uniform hashing*, that is, the keys are drawn from a distribution that makes them equally likely to go in any slot, independent of the other keys.
- We therefore have equal probabilities $\Pr(h(k) = x)$ for all x.
- If we are storing n keys, with m slots, what is the expected number of keys stored in a single slot?
- The expected number in each of the m slots is n/m.
- We write $\alpha = n/m$, and call it the *load factor*.
- If we are given a new key k, what is the expected time it takes to search to see if k is in the hash table?
- We will have to go through one of the slots to see if it is there.

Analysis

- Let's work with the assumption of *simple uniform hashing*, that is, the keys are drawn from a distribution that makes them equally likely to go in any slot, independent of the other keys.
- We therefore have equal probabilities $\Pr(h(k) = x)$ for all x.
- If we are storing n keys, with m slots, what is the expected number of keys stored in a single slot?
- The expected number in each of the m slots is n/m.
- We write $\alpha = n/m$, and call it the *load factor*.
- If we are given a new key k, what is the expected time it takes to search to see if k is in the hash table?
- We will have to go through one of the slots to see if it is there.
- Let's break into two cases, depending on whether it's in the table already or not.

Analysis

If we are given a new key k, what is the expected time it takes to search to see if k is in the hash table?

Analysis

If we are given a new key k, what is the expected time it takes to search to see if k is in the hash table?

- First, let's suppose that k is actually not in the hash table.

Analysis

If we are given a new key k, what is the expected time it takes to search to see if k is in the hash table?

- First, let's suppose that k is actually not in the hash table.
- We have to take $\Theta(1)$ time to compute $h(k)$ and access the slot.
- It is equally likely that k is hashed to any one of the m slots (that's the simple uniform hashing assumption).

Analysis

If we are given a new key k, what is the expected time it takes to search to see if k is in the hash table?

- First, let's suppose that k is actually not in the hash table.
- We have to take $\Theta(1)$ time to compute $h(k)$ and access the slot.
- It is equally likely that k is hashed to any one of the m slots (that's the simple uniform hashing assumption).
- The expected time is therefore:

$$\mathbb{E}[\text{total time}] = \Theta(1) + \sum_{i=1}^{m} \frac{1}{m} \mathbb{E}[\text{time searching slot } i]$$

Analysis

If we are given a new key k, what is the expected time it takes to search to see if k is in the hash table?

- First, let's suppose that k is actually not in the hash table.
- We have to take $\Theta(1)$ time to compute $h(k)$ and access the slot.
- It is equally likely that k is hashed to any one of the m slots (that's the simple uniform hashing assumption).
- The expected time is therefore:

$$\begin{aligned}\mathbb{E}[\text{total time}] &= \Theta(1) + \sum_{i=1}^{m} \frac{1}{m}\mathbb{E}[\text{time searching slot } i] \\ &= \Theta(1) + \sum_{i=1}^{m} \frac{1}{m} O(\alpha)\end{aligned}$$

Analysis

If we are given a new key k, what is the expected time it takes to search to see if k is in the hash table?

- First, let's suppose that k is actually not in the hash table.
- We have to take $\Theta(1)$ time to compute $h(k)$ and access the slot.
- It is equally likely that k is hashed to any one of the m slots (that's the simple uniform hashing assumption).
- The expected time is therefore:

$$\begin{aligned}\mathbb{E}[\text{total time}] &= \Theta(1) + \sum_{i=1}^{m} \frac{1}{m}\mathbb{E}[\text{time searching slot } i] \\ &= \Theta(1) + \sum_{i=1}^{m} \frac{1}{m} O(\alpha) \\ &= \Theta(1) + \Theta(\alpha)\end{aligned}$$

Analysis

If we are given a new key k, what is the expected time it takes to search to see if k is in the hash table?

- First, let's suppose that k is actually not in the hash table.
- We have to take $\Theta(1)$ time to compute $h(k)$ and access the slot.
- It is equally likely that k is hashed to any one of the m slots (that's the simple uniform hashing assumption).
- The expected time is therefore:

$$\begin{aligned}\mathbb{E}[\text{total time}] &= \Theta(1) + \sum_{i=1}^{m} \frac{1}{m}\mathbb{E}[\text{time searching slot } i] \\ &= \Theta(1) + \sum_{i=1}^{m} \frac{1}{m} O(\alpha) \\ &= \Theta(1) + \Theta(\alpha) \\ &= \Theta(1+\alpha),\end{aligned}$$

where we write $1+\alpha$ since we don't know if $\alpha = n/m$ is bigger or less than 1.

Analysis

If we are given a new key k, what is the expected time it takes to search to see if k is in the hash table?

Analysis

If we are given a new key k, what is the expected time it takes to search to see if k is in the hash table?

- Now, let's suppose that k is in the hash table.

Analysis

If we are given a new key k, what is the expected time it takes to search to see if k is in the hash table?

- Now, let's suppose that k is in the hash table.
- We assume it is equally likely to be any one of the keys in the table.

Analysis

If we are given a new key *k*, what is the expected time it takes to search to see if *k* is in the hash table?

- Now, let's suppose that *k* is in the hash table.
- We assume it is equally likely to be any one of the keys in the table.
- The analysis is a bit different, since a longer list is now more likely to be searched (since there are more keys leading to it).

Analysis

If we are given a new key k, what is the expected time it takes to search to see if k is in the hash table?

- Now, let's suppose that k is in the hash table.
- We assume it is equally likely to be any one of the keys in the table.
- The analysis is a bit different, since a longer list is now more likely to be searched (since there are more keys leading to it).
- Suppose that the hash table was created by inserting $k_1, k_2, \ldots, k_n$ in that order.

Analysis

If we are given a new key k, what is the expected time it takes to search to see if k is in the hash table?

- Now, let's suppose that k is in the hash table.
- We assume it is equally likely to be any one of the keys in the table.
- The analysis is a bit different, since a longer list is now more likely to be searched (since there are more keys leading to it).
- Suppose that the hash table was created by inserting $k_1, k_2, \ldots, k_n$ in that order.
- k might be any of these with equal probability.

Analysis

If we are given a new key k, what is the expected time it takes to search to see if k is in the hash table?

- Now, let's suppose that k is in the hash table.
- We assume it is equally likely to be any one of the keys in the table.
- The analysis is a bit different, since a longer list is now more likely to be searched (since there are more keys leading to it).
- Suppose that the hash table was created by inserting $k_1, k_2, \ldots, k_n$ in that order.
- k might be any of these with equal probability.
- Let X_{ij} be the indicator variable that $h(k_i) = h(k_j)$.

Analysis

If we are given a new key k, what is the expected time it takes to search to see if k is in the hash table?

- Now, let's suppose that k is in the hash table.
- We assume it is equally likely to be any one of the keys in the table.
- The analysis is a bit different, since a longer list is now more likely to be searched (since there are more keys leading to it).
- Suppose that the hash table was created by inserting $k_1, k_2, \ldots, k_n$ in that order.
- k might be any of these with equal probability.
- Let X_{ij} be the indicator variable that $h(k_i) = h(k_j)$.
- What is the expected value of the position of k_i within its slot?

Analysis

If we are given a new key k, what is the expected time it takes to search to see if k is in the hash table?

- Now, let's suppose that k is in the hash table.
- We assume it is equally likely to be any one of the keys in the table.
- The analysis is a bit different, since a longer list is now more likely to be searched (since there are more keys leading to it).
- Suppose that the hash table was created by inserting $k_1, k_2, \ldots, k_n$ in that order.
- k might be any of these with equal probability.
- Let X_{ij} be the indicator variable that $h(k_i) = h(k_j)$.
- What is the expected value of the position of k_i within its slot?
- The expected position is 1 plus the expected number of elements inserted after we inserted k_i:

$$1 + \mathbb{E}\left[\sum_{j=i+1}^{n} X_{ij}\right].$$

Analysis

Analysis

- The expected position of k_i in its slot is:

$$1 + \mathbb{E}\left[\sum_{j=i+1}^{n} X_{ij}\right].$$

Analysis

- The expected position of k_i in its slot is:

$$1 + \mathbb{E}\left[\sum_{j=i+1}^{n} X_{ij}\right].$$

- Since k could be any k_i with equal probability, the expected position of k in its slot is:

$$\frac{1}{n}\sum_{i=1}^{n}\left(1 + \mathbb{E}\left[\sum_{j=i+1}^{n} X_{ij}\right]\right)$$

Analysis

- The expected position of k_i in its slot is:

$$1 + \mathbb{E}\left[\sum_{j=i+1}^{n} X_{ij}\right].$$

- Since k could be any k_i with equal probability, the expected position of k in its slot is:

$$\frac{1}{n}\sum_{i=1}^{n}\left(1 + \mathbb{E}\left[\sum_{j=i+1}^{n} X_{ij}\right]\right) = 1 + \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathbb{E}[X_{ij}]$$

Analysis

- The expected position of k_i in its slot is:

$$1 + \mathbb{E}\left[\sum_{j=i+1}^{n} X_{ij}\right].$$

- Since k could be any k_i with equal probability, the expected position of k in its slot is:

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{n}\left(1 + \mathbb{E}\left[\sum_{j=i+1}^{n} X_{ij}\right]\right) &= 1 + \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathbb{E}[X_{ij}] \\ &= 1 + \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{1}{m}\end{aligned}$$

Analysis

- The expected position of k_i in its slot is:

$$1 + \mathbb{E}\left[\sum_{j=i+1}^{n} X_{ij}\right].$$

- Since k could be any k_i with equal probability, the expected position of k in its slot is:

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{n}\left(1 + \mathbb{E}\left[\sum_{j=i+1}^{n} X_{ij}\right]\right) &= 1 + \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathbb{E}[X_{ij}] \\ &= 1 + \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{1}{m} = 1 + \frac{1}{nm}\sum_{i=1}^{n}(n-i)\end{aligned}$$

Analysis

- The expected position of k_i in its slot is:

$$1 + \mathbb{E}\left[\sum_{j=i+1}^{n} X_{ij}\right].$$

- Since k could be any k_i with equal probability, the expected position of k in its slot is:

$$\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n}\left(1 + \mathbb{E}\left[\sum_{j=i+1}^{n} X_{ij}\right]\right) &= 1 + \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathbb{E}[X_{ij}] \\
&= 1 + \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{1}{m} = 1 + \frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
&= 1 + \frac{1}{nm}((n-1) + (n-2) + \ldots + 1)
\end{aligned}$$

Analysis

- The expected position of k_i in its slot is:

$$1 + \mathbb{E}\left[\sum_{j=i+1}^{n} X_{ij}\right].$$

- Since k could be any k_i with equal probability, the expected position of k in its slot is:

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{n}\left(1 + \mathbb{E}\left[\sum_{j=i+1}^{n} X_{ij}\right]\right) &= 1 + \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathbb{E}[X_{ij}] \\ &= 1 + \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{1}{m} = 1 + \frac{1}{nm}\sum_{i=1}^{n}(n-i) \\ &= 1 + \frac{1}{nm}((n-1) + (n-2) + \ldots + 1) \\ &= 1 + \frac{1}{nm}\left(\frac{(n-1)(n)}{2}\right)\end{aligned}$$

Analysis

- The expected position of k_i in its slot is:

$$1 + \mathbb{E}\left[\sum_{j=i+1}^{n} X_{ij}\right].$$

- Since k could be any k_i with equal probability, the expected position of k in its slot is:

$$\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n}\left(1 + \mathbb{E}\left[\sum_{j=i+1}^{n} X_{ij}\right]\right) &= 1 + \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathbb{E}[X_{ij}] \\
&= 1 + \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{1}{m} = 1 + \frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
&= 1 + \frac{1}{nm}((n-1) + (n-2) + \ldots + 1) \\
&= 1 + \frac{1}{nm}\left(\frac{(n-1)(n)}{2}\right) = 1 + \frac{n-1}{2m}
\end{aligned}$$

Analysis

- The expected position of k_i in its slot is:

$$1 + \mathbb{E}\left[\sum_{j=i+1}^{n} X_{ij}\right].$$

- Since k could be any k_i with equal probability, the expected position of k in its slot is:

$$\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n}\left(1 + \mathbb{E}\left[\sum_{j=i+1}^{n} X_{ij}\right]\right) &= 1 + \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathbb{E}[X_{ij}] \\
&= 1 + \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{1}{m} = 1 + \frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
&= 1 + \frac{1}{nm}((n-1) + (n-2) + \ldots + 1) \\
&= 1 + \frac{1}{nm}\left(\frac{(n-1)(n)}{2}\right) = 1 + \frac{n-1}{2m}
\end{aligned}$$

- Since $n/m = \alpha$, this is $\Theta(1+\alpha)$, so the total time to search for k (including $\Theta(1)$ for computing $h(k)$) is $\Theta(1) + \Theta(1+\alpha) = \Theta(1+\alpha)$.

Simple hashing methods

Simple hashing methods

- If we know that keys k are real numbers with the uniform distribution on the interval $[0, 1]$, can do:
  $$h(k) = \lfloor km \rfloor,$$
  where $\lfloor x \rfloor$ is the *floor function* that rounds down to the nearest integer.

Simple hashing methods

- If we know that keys k are real numbers with the uniform distribution on the interval $[0, 1]$, can do:
  $$h(k) = \lfloor km \rfloor,$$
  where $\lfloor x \rfloor$ is the *floor function* that rounds down to the nearest integer.
- This gives equal probability of $h(k) = 0, 1, \ldots, m - 1$.

Simple hashing methods

- If we know that keys k are real numbers with the uniform distribution on the interval $[0, 1]$, can do:
$$h(k) = \lfloor km \rfloor,$$
where $\lfloor x \rfloor$ is the *floor function* that rounds down to the nearest integer.
- This gives equal probability of $h(k) = 0, 1, \ldots, m - 1$.
- If k are integers, we can use residues modulo m:
$$h(k) = k \pmod{m}.$$

Simple hashing methods

- If we know that keys k are real numbers with the uniform distribution on the interval $[0, 1]$, can do:

  $$h(k) = \lfloor km \rfloor,$$

  where $\lfloor x \rfloor$ is the *floor function* that rounds down to the nearest integer.
- This gives equal probability of $h(k) = 0, 1, \ldots, m - 1$.
- If k are integers, we can use residues modulo m:

  $$h(k) = k \pmod{m}.$$

- This gives again $h(k) = 0, 1, \ldots, m - 1$, though need to know that keys are equally likely to have different residues.

Simple hashing methods

- If we know that keys k are real numbers with the uniform distribution on the interval $[0,1]$, can do:

  $$h(k) = \lfloor km \rfloor,$$

  where $\lfloor x \rfloor$ is the *floor function* that rounds down to the nearest integer.
- This gives equal probability of $h(k) = 0, 1, \ldots, m-1$.
- If k are integers, we can use residues modulo m:

  $$h(k) = k \pmod{m}.$$

- This gives again $h(k) = 0, 1, \ldots, m-1$, though need to know that keys are equally likely to have different residues.
- In practice, we would want to avoid certain m, in particular powers of two, since often residues mod 2^s have patterns for certain data.

Simple hashing methods

- If we know that keys k are real numbers with the uniform distribution on the interval $[0, 1]$, can do:

  $$h(k) = \lfloor km \rfloor,$$

  where $\lfloor x \rfloor$ is the *floor function* that rounds down to the nearest integer.
- This gives equal probability of $h(k) = 0, 1, \ldots, m - 1$.
- If k are integers, we can use residues modulo m:

  $$h(k) = k \pmod{m}.$$

- This gives again $h(k) = 0, 1, \ldots, m - 1$, though need to know that keys are equally likely to have different residues.
- In practice, we would want to avoid certain m, in particular powers of two, since often residues mod 2^s have patterns for certain data.
- A prime number m is often good.

Universal hashing

Universal hashing

- We have a problem if an adversary gets to pick the keys for either of the above hash functions.

Universal hashing

- We have a problem if an adversary gets to pick the keys for either of the above hash functions.
- Can make them all collide very easily.

Universal hashing

- We have a problem if an adversary gets to pick the keys for either of the above hash functions.
- Can make them all collide very easily.
- In *universal hashing*, the hash function itself is picked randomly to avoid this.

Universal hashing

- We have a problem if an adversary gets to pick the keys for either of the above hash functions.
- Can make them all collide very easily.
- In *universal hashing*, the hash function itself is picked randomly to avoid this.
- Specifically, the goal is that for *any* pair of keys (even not random), we are likely to pick a hash function where they don't collide.

Universal hashing

- We have a problem if an adversary gets to pick the keys for either of the above hash functions.
- Can make them all collide very easily.
- In *universal hashing*, the hash function itself is picked randomly to avoid this.
- Specifically, the goal is that for *any* pair of keys (even not random), we are likely to pick a hash function where they don't collide.
- It's basically moving the randomness assumption from the keys to the hash function, since we know we can control that.

Universal hashing

- We have a problem if an adversary gets to pick the keys for either of the above hash functions.
- Can make them all collide very easily.
- In *universal hashing*, the hash function itself is picked randomly to avoid this.
- Specifically, the goal is that for *any* pair of keys (even not random), we are likely to pick a hash function where they don't collide.
- It's basically moving the randomness assumption from the keys to the hash function, since we know we can control that.
- Formally, let $\mathcal{H}$ be a set of possible hash functions, from which we pick an h at random.

Universal hashing

- We have a problem if an adversary gets to pick the keys for either of the above hash functions.
- Can make them all collide very easily.
- In *universal hashing*, the hash function itself is picked randomly to avoid this.
- Specifically, the goal is that for *any* pair of keys (even not random), we are likely to pick a hash function where they don't collide.
- It's basically moving the randomness assumption from the keys to the hash function, since we know we can control that.
- Formally, let $\mathcal{H}$ be a set of possible hash functions, from which we pick an h at random.
- We say that $\mathcal{H}$ is *universal* if for any two different keys k_1, k_2, the probability of picking h with $h(k_1) = h(k_2)$ is at most $1/m$.

Universal hashing

Universal hashing

- Let’s look at one way to design a universal set of hash functions.

Universal hashing

- Let’s look at one way to design a universal set of hash functions.
- Assume that keys are integers in a specified range.

Universal hashing

- Let's look at one way to design a universal set of hash functions.
- Assume that keys are integers in a specified range.
- Pick a prime p so every k is in the range 0 to $p - 1$.

Universal hashing

- Let's look at one way to design a universal set of hash functions.
- Assume that keys are integers in a specified range.
- Pick a prime p so every k is in the range 0 to $p - 1$.
- Since we assume more keys than slots, we have $p > m$.

Universal hashing

- Let's look at one way to design a universal set of hash functions.
- Assume that keys are integers in a specified range.
- Pick a prime p so every k is in the range 0 to $p - 1$.
- Since we assume more keys than slots, we have $p > m$.
- For every $a \in \{1, 2, \ldots, p-1\}$ and $b \in \{0, 1, 2, \ldots, p-1\}$, we define:

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

Universal hashing

- Let's look at one way to design a universal set of hash functions.
- Assume that keys are integers in a specified range.
- Pick a prime p so every k is in the range 0 to $p-1$.
- Since we assume more keys than slots, we have $p > m$.
- For every $a \in \{1, 2, \ldots, p-1\}$ and $b \in \{0, 1, 2, \ldots, p-1\}$, we define:

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

- How many choices of h_{ab} are there?

Universal hashing

- Let's look at one way to design a universal set of hash functions.
- Assume that keys are integers in a specified range.
- Pick a prime p so every k is in the range 0 to $p - 1$.
- Since we assume more keys than slots, we have $p > m$.
- For every $a \in \{1, 2, \ldots, p - 1\}$ and $b \in \{0, 1, 2, \ldots, p - 1\}$, we define:

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

- How many choices of h_{ab} are there?
- $p - 1$ choices for a and p for b, so $p(p - 1)$ choices for h_{ab}.

Universal hashing

- Let's look at one way to design a universal set of hash functions.
- Assume that keys are integers in a specified range.
- Pick a prime p so every k is in the range 0 to $p - 1$.
- Since we assume more keys than slots, we have $p > m$.
- For every $a \in \{1, 2, \ldots, p-1\}$ and $b \in \{0, 1, 2, \ldots, p-1\}$, we define:

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

- How many choices of h_{ab} are there?
- $p - 1$ choices for a and p for b, so $p(p - 1)$ choices for h_{ab}.
- We will prove that this set of hash functions is universal.