Agenda:

- Stacks & Queues (CLRS Ch 10.1)
- Basics of Amortized Analysis (CLRS Ch 17.1)

- Hash-Tables (CLRS Ch 11)

Stacks and Queues:

- Two highly used data-structures
- Supporting `Insert` and `Remove` in $O(1)$ time
- Implementing LIFO/FIFO order
  - Stacks: Implement Last-In-First-Out order
  - Queues: Implement First-In-First-Out order

Hash-Table:

- Another highly used data-structure
- Supports `Insert`, `Remove` and `Find` in $O(1)$ time
- But keeps items in no particular order.

LIFO/FIFO:

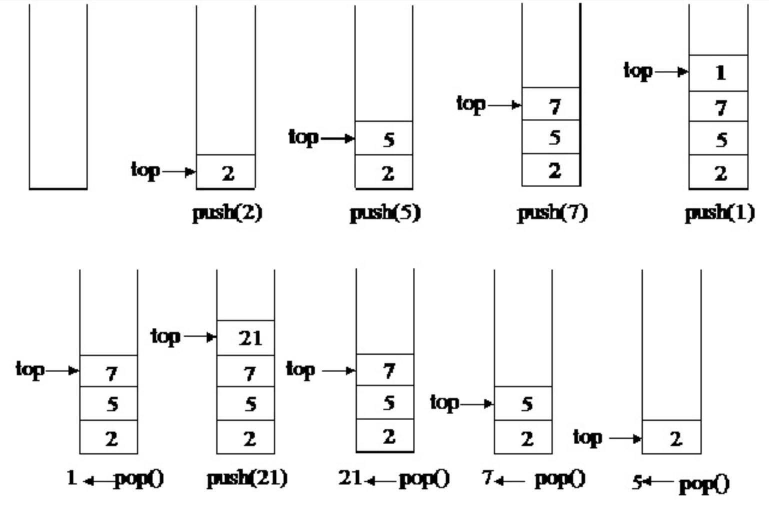
- An example:
  `Insert(4), Insert(17), Insert(33), Remove(), Remove(), Insert(12), Remove(), insert(35), Remove(), Remove()`
- Note how `Remove()` doesn't call a particular element — it is basically "bring me the next element"
- On a Queue, the order of items as they are taken out of the queue: 4, 17, 33, 12, 35
- On a Stack, the order of items as they are taken out of the stack: 33, 17, 12, 35, 4

- On a Queue — `Insert` and `Remove` are referred to as `Enqueue` and `Dequeue`
- On a Stack — `Insert` and `Remove` are referred to as `Push` and `Pop`

Implementing a Stack with an Array:

- Use an array $A[1, \ldots n]$ of capacity n,
  and a pointer top to the last cell in A where we placed an element.
- `Initialize`(n)
  Create an array A of size n
  $capacity \leftarrow n$
  $top \leftarrow 0$
- `Push`(x)
  `if` $(top < capacity)$
  $\quad top \leftarrow top + 1$
  $\quad A[top] \leftarrow x$
- `Pop`$()$
  `if` $(top > 0)$
  $\quad key \leftarrow A[top]$
  $\quad top \leftarrow top - 1$
  $\quad$ `return` key
- Clearly, runtime of `Push` and `Pop` is $O(1)$.

Implementing a Stack with an Array:

- 

Implementing a Queue a Doubly-Linked List

- Very easy:
  - `Enqueue()` — insert a new list head
  - `Dequeue()` — remove the list's tail

Implementing a Queue with an Array:

- Attempt #1: Uses an array $A[1, ...n]$ of capacity n, and a pointer to the last element we put in the queue.
- What will we do upon `Dequeue()`?
- Return $A[1]$.
- But now we need to move all elements in $A[2, ..., last]$ to $A[1, ..., last - 1]$. This takes too much time...
- Instead, we keep a pointer to the end of the queue (where we placed the last element) and a point to the front of the queue (where we placed the first element).
  We will pretend the array is cyclic: when we advance a pointer by 1, if its value was n then its new value is set to 1.

Implementing a Queue with an Array:

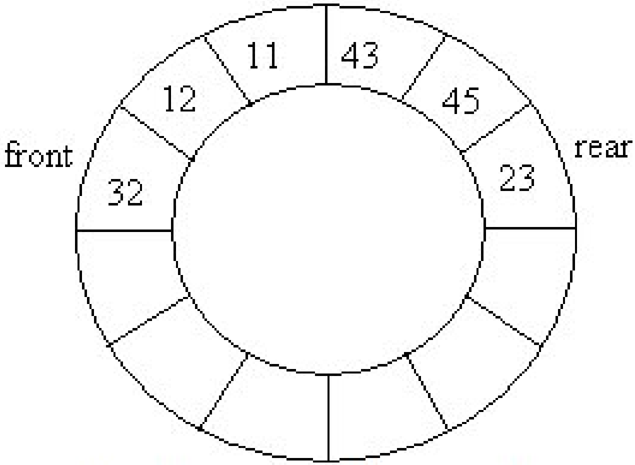
- Uses an array $A[1, \ldots n]$ of capacity n and keeps track of its current $size$, and two pointers: $front$ to the head of the queue, $back$ to the tail of the queue.
- `Initialize`(n)
  Create an array A of size n
  $capacity \leftarrow n$
  $size \leftarrow 0$
  $front \leftarrow 1$
  $back \leftarrow 0$

- `Enqueue`(x)
  if $(size < capacity)$
      $back \leftarrow back + 1$
      if $(back = n + 1)$
          $back = 1$
      $A[back] = x$
      $size \leftarrow size + 1$

- `Dequeue`$()$
  if $(size > 0)$
      $key \leftarrow A[front]$
      $front \leftarrow front + 1$
      if $(front = n + 1)$
          $front = 1$
      $size \leftarrow size - 1$
      return key

- Clearly, runtime of `Enqueue` and `Dequeue` is $O(1)$.

Implementing a Queue with an Array:

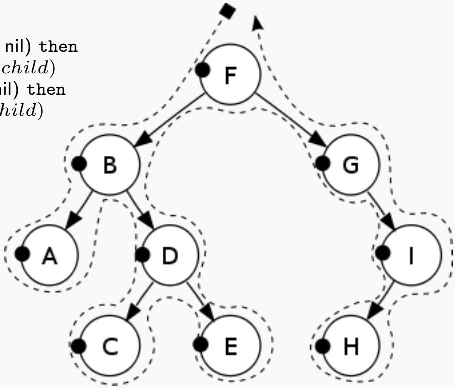


Circular queue with some values

▶

An Example for Using a Stack and a Queue:

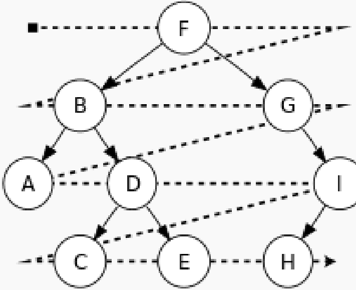
- Print the nodes of a tree in pre-order
- <u>`procedure Pre-order`$(root)$</u>
  `Stack_Initialize`(S)
  `Push`$(S, root)$
  `while` (S isn't empty)
      $v \leftarrow$ `Pop`(S)
      `Print`(v)
      `if`$(v.rightchild \neq$ nil$)$ `then`
          `Push`$(S, v.rightchild)$
      `if`$(v.leftchild \neq$ nil$)$ `then`
          `Push`$(S, v.leftchild)$



-

An Example for Using a Stack and a Queue:

- Print the nodes of a tree by level
- Procedure Level-order($root$)
  Queue_Initialize(S)
  Enqueue($S, root$)
  Enqueue($S, \perp$) $\quad **\ \perp =$a special sign meaning "new line"
  while (S isn't empty)
      $v \leftarrow$ Dequeue(S)
      if ($v \neq \perp$) then
          Print(v)
          if($v.leftchild \neq$nil) then
              Enqueue($S, v.leftchild$)
          if($v.rightchild \neq$nil) then
              Enqueue($S, v.rightchild$)
      else
          Print New Line
          if (S isn't empty)
              Enqueue($S, \perp$)



-

Must Queues, Stacks and Hash-Tables have a capacity?

- Sometimes we know a-priori a bound on how many elements will populate the Stack/Queue at any given time...
- And sometimes we don't...
- Here's one solution once we've inserted n elements and already at full-capacity:
    1. Create a Stack / Queue / Table of size $2\times$current-capacity
    2. Copy all elements / re-insert all n elements to the new data-structure
    3. Delete original data-structure.
- Runtime $= \Theta(n)$ (we're copying n elements / or doing n insertion operations)
- So now, `Push()`, `Enqueue()` or `HashTable_Insert()` take $O(n)$ time in the worst-case...
- Does that mean that if we do n `Push()` operations our overall runtime is $O(n^2)$?

Amortized Analysis

- Let's see an example: $n = 4$.
- `Push(1), Push(2), Push(3), Push(4)`
  - … so far, 4 operations, each one in $O(1)$ time
- `Push(5)`
  - Oops… now we need to do (something proportional to) 4 steps.
  - And then push in 5 — 1 more extra step
- `Push(6)`
  - Only $O(1)$…
- `Push(7), Push(8)`
  - Only $O(1)$ per each step…
- `Push(9)`
  - We now make (something proportional to) 8 steps…
- `Push(10)`
  - Only $O(1)$…
- `Push(11), Push(12), Push(13), Push(14), Push(15), Push(16)`
  - Only $O(1)$ per each step.
- To insert 16 elements, we make $4 + 4 + 4 + 8 + 8 = 28$ steps.
  But to insert 17 elements, we make $28 + 16 + 1 = 45$ steps.
- To insert 32 elements, we make $44 + 16 = 60$ steps.
  But to insert 33 elements, we make $60 + 32 + 1 = 93$ steps.

Amortized Analysis

- In any sequence of `Push()`, `Pop()`, before making a call for `Push()` that invoked n steps, we had to make *at least* $n/2$ steps where pushing/popping cost us just $O(1)$.
- Denote the overall runtime of n `Push()`, `Pop()` operations as $T(n)$.
- Then we just inferred that $T(n) \leq 3n$.
- You can prove this by induction.
  Base case: $T(1) = 1 \leq 3$.
  Induction step: If the last action took us just 1 step, then
  $T(n) = T(n-1) + 1 \leq 3n - 3 + 1 \leq 3n$.
  Otherwise, the last action took us n steps, which means all $\frac{n}{2}$ actions before it took only one step. Therefore:

$$T(n) = T(\tfrac{n}{2}) + \tfrac{n}{2} + n = \tfrac{3n}{2} + \tfrac{n}{2} + n = 3n$$

- Therefore, our **amortized cost**, defined as $\frac{T(n)}{n} = O(1)$.
- The subject of Amortized Analysis is deep and beautiful, and I encourage you to read more at the CLRS book. (Chapter 17)

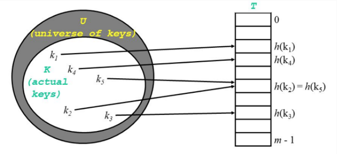
Hash-Table:

- A data structure
- Supports `insert(`key`)`, `find(`key`)` and `remove(`key`)` — in $O(1)$.
- Doesn't keep the keys sorted.

- When is it useful?
- When there's a large universe U of potential keys, but we use only n keys.
  - E.g., think of the warehouse of "Ramazon," a new online shopping website. You type in a product's serial number and it checks whether the product exists in the Ramazon's warehouse.
  - E.g., student IDs for UofA — I just want to access the record of student #4413928
  - E.g., think of your C compiler: the universe of potential names for variables and functions is huge — but your code only have a moderate amount of variables and function names.

Hash-Table:

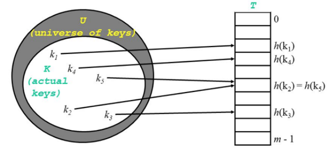
- A data structure: `insert`(key), `find`(key) and `remove`(key) — in $O(1)$.
- The main idea: use a easy-to-compute Hash function that maps $h : U \to \{1, 2..., m\}$. (Ideally $m = n$, but maybe we have $m = O(n)$.)



- Since multiple keys may be mapped to the same value $h(key_1) = h(key_2) = v$ then $T[v]$ stores a list / an array of elements.
- So `insert`(key), `find`(key) and `remove`(key) all work by
    1. Compute $v = h(key)$
    2. Goto $T[v]$ (in $O(1)$)
    3. Traverse the list in $T[v]$ to add/search/remove
- Runtime: $O(\text{computing } h(key)) + O(\text{length of list in } T[v])$
- We use a simple h, so assume computing $h(key)$ takes $O(1)$
  The dominating factor is the length of the longest list in T
- The ideal scenario: h is injective — makes all lists be of length 1.
- Since we assume $|U| \gg m$ we can't have an injective h.
- In fact, worst-case — all n elements have the same hash-value and the table is reduced to a linked-list...

Hash-Table:

- A data structure: `insert`(key), `find`(key) and `remove`(key) — in $O(1)$.
- The main idea: use a easy-to-compute Hash function that maps $h : U \to \{1, 2..., m\}$. (Ideally $m = n$, but maybe we have $m = O(n)$.)



- The ideal scenario: h is injective **for our n keys**.
- How to design an injective h for an unknown set of keys???
- Answer: use randomness.
- h that works well in practice:
  - map $h(k) = (k \mod m)$
  - If distribution of keys is uniform over $\{1, 2, ..., |U|\}$, then distribution of $h(k)$s is ~uniform on $\{0, 1, 2, ..., m - 1\}$.
- An even better h that comes with provable guarantees:
  - Pick some prime $p > |U|$
  - Pick a uniformly at random from $\{1, 2, ..., p - 1\}$ and b uniformly at random from $\{0, 1, 2, .., p - 1\}$
  - Map $h(k) = \Big((a \cdot k + b) \mod p\Big) \mod m$

Operations Modulo Prime p

- Given any integer a we can write a in a unique way as $a = k \times p + r$ where k is an integer and r is an integer in $\{0, 1, 2, ..., p-1\}$.
- And so $a \mod p = r$.
  - For any integer a, even negatives: $-1 = (-1) \times p + (p-1)$
- Addition Modulo p: For any integers a, b we have

$$(a+b) \mod p = \big((a \mod p) + (b \mod p)\big) \mod p$$

  - Write $a = k \times p + r, b = l \times p + s$.
  - Then $a + b = (k+l)p + (r+s)$. So, $(a+b) \mod p = (r+s) \mod p$.
- Multiplication Modulo p: For any integers a, b we have

$$(a \times b) \mod p = \big((a \mod p) \times (b \mod p)\big) \mod p$$

  - Write $a = k \times p + r, b = l \times p + s$.
  - Then $a \times b = (kl + ks + rl)p + rs$. Hence, $(a \times b) \mod p = (r \times s) \mod p$.
- Both of these properties doesn't require p to be prime. It can be any integer.
- Here's a property that is true only for prime numbers:
  Fermat's Little Thm: For any integer a that is not divisible by p we have that $a^{p-1} \mod p = 1$.

Operations Modulo Prime p

- Fermat's Little Thm: For any integer a that is not divisible by p we have that $a^{p-1} \mod p = 1$
- Corollary: any a that is not divisible by p has a multiplicative inverse: $a^{-1} \stackrel{\text{def}}{=} a^{p-2} \mod p$

$$(a \times a^{-1}) \mod p = (a \times a^{p-2}) \mod p = a^{p-1} \mod p = 1$$

- OPTIONAL: The proof of Fermat's Little Thm is based on the following claim.
  Claim: For any integer $0 \leq x \leq p-1$ we have $x^p \mod p = x$.
- Assuming the claim is true, we prove the theorem.
- Fix any a not-divisible by p. Then

$$a^p \mod p \stackrel{\text{multiplication rule}}{=} (a \mod p)^p \mod p \stackrel{\text{claim}}{=} a \mod p$$

  because $(a \mod p)$ is a non-zero integer smaller than p.
- Hence:

$$0 = (a \mod p)^p - (a \mod p) = (a \mod p)\Big((a \mod p)^{p-1} - 1\Big) \mod p$$

  and since we know $(a \mod p) \neq 0$ then it must be the case that

$$\Big((a \mod p)^{p-1} - 1\Big) \mod p = 0$$

  which means $1 = (a \mod p)^{p-1} \mod p = a^{p-1} \mod p.$ ■

Operations Modulo Prime p

- Here's a property that is true only for prime numbers:
  Fermat's Little Thm: For any integer a such that $a \mod p \neq 0$ we have $a^{p-1} \mod p = 1$.
- OPTIONAL:
- Claim: For any integer $0 \leq x \leq p-1$ we have $x^p \mod p = x$.
- We prove the claim inductively. Clearly, it is true for $x = 0$ as $0^{p-1} = 0$ even without modulo operations.
- Fix $0 \leq n \leq p-2$. Assuming the claim holds for n we show it also holds for $n+1$.
- Note: $(n+1)^p \mod p = \sum_{j=0}^{p} \binom{p}{j} n^j$.
- For any $1 \leq j \leq p-1$ we have $\binom{p}{j} = \frac{p!}{j!(p-j)!}$. Note that all terms in the denominator are multiplications of integers strictly smaller than p, whereas we have p in the numerator. So p divided the numerator, but not the denominator.
  Since p is prime, and divisible only by itself, then the p term in the numerator isn't canceled by any term in the denominator.
  Hence $\binom{p}{j}$ is divisible by p.
- This leaves us with $(n+1)^p \mod p = (n^p \mod p) + (1^p \mod p) \stackrel{\text{IH}}{=} (n \mod p) + 1 = (n+1) \mod p.$ ■

Hash-Table:

- An h that comes with provable guarantees:
  - Pick some prime $p > |U|$
  - Pick a u.a.r from $\{1, 2, ..., p-1\}$ and b u.a.r from $\{0, 1, .., p-1\}$
  - Map $h(k) = \Big((a \cdot k + b) \mod p\Big) \mod m$
- Claim: Given n arbitrary distinct keys $k_1, .., k_n$, then for each key the expected length of its list is at most $1 + \frac{n}{m}(1 + o(1))$.
- Note: the keys are chosen adversarially, the expectation is over our choice of a and b.
- Proof: We will prove the following lemma:
  - Main Lemma: For any two distinct keys $k \neq k'$ we have that $\Pr_{a,b}[h(k) = h(k')] \leq \frac{1}{m} + \frac{1}{p}$
- Given the lemma, the proof of the claim uses linearity of expectation.
  - For any two distinct keys $k_i \neq k_j$ let $Y_{i,j}$ be the Bernoulli random variable indicating whether there's a collision of key k_i, k_j: $Y_{i,j} = 1$ iff $h(k_i) = h(k_j)$.
  - Thus, for each i, denote $n_i \stackrel{\text{def}}{=}$ length of the list k_i is in $=$ number of keys in $\{k_1, .., k_n\}$ that have $h(k_j) = h(k_i)$.
  - In other words, $n_i = \sum_j Y_{i,j} = 1 + \sum_{j \neq i} Y_{i,j}$. Hence, for any i we have

$$\mathrm{E}[n_i] = \mathrm{E}\left[1 + \sum_{j \neq i} Y_{i,j}\right] = 1 + \sum_{j \neq i} \mathrm{E}[Y_{i,j}] = 1 + \sum_{j \neq i} 1 \cdot \Pr[h(k_i) = h(k_j)]$$

$$\leq 1 + (n-1) \cdot (\tfrac{1}{m} + \tfrac{1}{p}) = 1 + \frac{n-1}{m} + \frac{n-1}{p} = 1 + \frac{n-1}{m}\left(1 + \frac{m}{p}\right) \quad \square$$

Hash-Table: Proof of Main Lemma (OPTIONAL)

- An h that comes with provable guarantees:
  - Pick some prime $p > |U|$
  - Pick a u.a.r from $\{1, 2, ..., p-1\}$ and b u.a.r from $\{0, 1, .., p-1\}$
  - Map $h(k) = \Big((a \cdot k + b) \mod p\Big) \mod m$
- Lemma: For any two distinct keys $k \neq k'$ we have that $\Pr_{a,b}[h(k) = h(k')] \leq \frac{1}{m} + \frac{1}{p}$.

- Proof: Denote $w(k) = (a \cdot k + b) \mod p$. (I.e., $h(k) = w(k) \mod m$.)
- First, we note that for any $k \neq k'$ we have $w(k) \neq w(k')$.
  - Suppose you are given two keys k and k' such that $w(k) = w(k')$, namely that $(a \cdot k + b) \mod p = (a \cdot k' + b) \mod p$.
  - Arithmetic manipulation (adding $-b$) gives that $a \cdot (k - k') \mod p = 0$.
  - The remainder is 0, which means p *divides* $a \cdot (k - k')$. We denote it as $p \mid a \cdot (k - k')$.
  - p divides the multiplication $p \mid a(k - k')$, but p is prime, so either $p \mid a$ or $p \mid (k - k')$.
  - Note that $a \neq 0$ and $a < p$, so $p \nmid a$.
  - Note also that both $0 \leq k, k' \leq |U| < p$ which means $-p < k - k' < p$.
  - So we must have $k - k' = 0$, hence $k = k'$.

Hash-Table: Proof of Main Lemma (OPTIONAL)

- An h that comes with provable guarantees:
  - Pick some prime $p > |U|$
  - Pick a u.a.r from $\{1, 2, ..., p-1\}$ and b u.a.r from $\{0, 1, .., p-1\}$
  - Map $h(k) = \Big((a \cdot k + b) \mod p\Big) \mod m$
- Lemma: For any two distinct keys $k \neq k'$ we have that $\Pr_{a,b}[h(k) = h(k')] \leq \frac{1}{m} + \frac{1}{p}$.

- Proof: (Cont'd) Denote $w(k) = (a \cdot k + b) \mod p$.
- Secondly, we argue the for any $k \neq k'$ and any integers $i \neq j$ which are smaller than p, we have that $\Pr_{a,b}[w(k) = i \text{ and } w(k') = j] = \frac{1}{p(p-1)}$.
  - $w(k) = i$ and $w(k') = j$ mean that $i = (a \cdot k + b) \mod p$ and $j = (a \cdot k' + b) \mod p$.
  - Hence, $(i - j) \mod p = (a \cdot (k - k')) \mod p$.
  - Because p is prime and $k - k' \neq 0$, the multiplicative inverse $(k - k')^{-1}$ exists.
  - Multiplying both sides by $(k - k)^{-1}$ we get: $a = a \mod p = \big((i - j) \cdot (k - k')^{-1}\big) \mod p$

    We thus denote $x \stackrel{\text{def}}{=} \big((i - j) \cdot (k - k')^{-1}\big) \mod p$ (some specific non-zero integer). We have shown that a must take the value x.
  - And: $b = (i - a \cdot k) \mod p = (i - x \cdot k) \mod p$.

    So b must take the value $y \stackrel{\text{def}}{=} (i - x \cdot k) \mod p$.
  - You can check that the inverse also holds: if $a = x$ and $b = y$ then $w(k) = i$ and $w(k') = j$.
  - Conclusion: $\Pr[w(k) = i \text{ and } w(k') = j] = \Pr[a = x \text{ and } b = y] = \frac{1}{p-1} \cdot \frac{1}{p}$

Hash-Table: Proof of Main Lemma (OPTIONAL)

- An h that comes with provable guarantees:
  - Pick some prime $p > |U|$
  - Pick a u.a.r from $\{1, 2, ..., p-1\}$ and b u.a.r from $\{0, 1, .., p-1\}$
  - Map $h(k) = \Big((a \cdot k + b) \mod p\Big) \mod m$
- Lemma: For any two distinct keys $k \neq k'$ we have that $\Pr_{a,b}[h(k) = h(k')] \leq \frac{1}{m} + \frac{1}{p}$.

- Proof: (Cont'd.) We now argue the following.
- For every $i < m$, the set $S_i = \{0 \leq x < p : \ x \mod m = i\}$ has size $\leq 1 + \lfloor \frac{p}{m} \rfloor$.
- Note that the set S_i holds precisely all the elements that $= i(\mod m)$.
  - Partition the integers between 0 and $p-1$ into consecutive chunks, each of size m (except for the last one): $\{0, 1, .., m-1\}, \{m, m+1, m+2, ..., 2m-1\}, ..., \{m \cdot \lfloor \frac{p}{m} \rfloor, ..., p-1\}$
  - In each chuck exactly 1 element has remainder of i when divided by m, except for the last chunk which either have such an element or has no such elements.
  - There are $\lceil \frac{p}{m} \rceil = 1 + \lfloor \frac{p}{m} \rfloor = 1 + \lfloor \frac{p-1}{m} \rfloor$ such chunks (remember, p is prime, so not divisible by m).
  - Thus, $|S_i| \leq$#chunks$= 1 + \lfloor \frac{p}{m} \rfloor = 1 + \lfloor \frac{p-1}{m} \rfloor$.

Hash-Table: Proof of Main Lemma (OPTIONAL)

- An h that comes with provable guarantees:
  - Pick some prime $p > |U|$
  - Pick a u.a.r from $\{1, 2, ..., p-1\}$ and b u.a.r from $\{0, 1, .., p-1\}$
  - Map $h(k) = \Big((a \cdot k + b) \mod p\Big) \mod m$
- Lemma: For any two distinct keys $k \neq k'$ we have that $\Pr_{a,b}[h(k) = h(k')] \leq \frac{1}{m} + \frac{1}{p}$.

- Proof: (Cont'd.) We can now conclude the proof of the lemma.
- For any distinct $k \neq k'$, $\Pr[h(k) = h(k')] \leq (\frac{1}{m} + \frac{1}{p})$, because:

$$\begin{aligned}
\Pr[h(k) = h(k')] &= \sum_{i=0}^{m-1} \Pr[h(k) = h(k') = i] \\
&= \sum\nolimits_i \Pr[w(k) \in S_i \text{ and } w(k') \in S_i \setminus \{w(k)\}] \\
&= \sum\nolimits_i \frac{|S_i|(|S_i| - 1)}{p(p-1)} \leq \sum_i \frac{(1 + \lfloor \frac{p}{m} \rfloor) \cdot \lfloor \frac{p}{m} \rfloor}{p(p-1)} \\
&= m \cdot \frac{1 + \lfloor \frac{p}{m} \rfloor}{p} \cdot \frac{\lfloor \frac{p-1}{m} \rfloor}{p-1} \leq m \cdot \frac{1 + \frac{p}{m}}{p} \cdot \frac{\frac{p-1}{m}}{p-1} \\
&= m \cdot (\tfrac{1}{p} + \tfrac{1}{m}) \cdot \tfrac{1}{m} = \tfrac{1}{m} + \tfrac{1}{p} \quad \square
\end{aligned}$$

Hash-Table

- The proof we gave basically shows the following.
  - We call a set of function $\mathcal{H}$ a *universal hash family* if the following holds. For any two distinct keys $k \neq k'$, when picking h from $\mathcal{H}$ u.a.r we have
  $$\Pr_{h\sim\mathcal{H}}[h(k) = h(k')] = \frac{1}{m}$$
  - You can check and see that we have effectively shown that
    1. If $\mathcal{H}$ is a universal hash family, then by picking a function h from $\mathcal{H}$ u.a.r the expected length of each list is at most $1 + \frac{n}{m}$
    (This is the proof of the claim based on the lemma.)
    2. The family $\mathcal{H}_{a,b} = \{h_{a,b}(k) = (a \cdot k + b) \mod m\}$ that has $p(p-1)$ elements, is a universal hash family.[1] (This was the proof of the main lemma.)
- What we have shown is that for every key, $\mathrm{E}[\#\text{collisions}] \approx 1 + \frac{n}{m}$.
- Hence, $\max_j \mathrm{E}[\#\text{collisions with } k_j] \approx 1 + \frac{n}{m}$.
- But what we'd love to bound is $\mathrm{E}[\max_j \{\#\text{collisions with } k_j\}]$ (why?) That's much more difficult.
  - Even if h is completely random ($\forall k, h(k)$ is distributed uniformly among the m bins) it is known that for $m = n$ the expected size of the largest bin is about $O(\log(n))$.
- Read more on hash functions, universal hashing and open addressing hashing — Chapter 11

[1] up to the little "fudge" factor of $\frac{1}{p}$.