Consistent hashing:

- There are $\boxed{m \text{ items}}$ such that each of them needs to be stored in one of the $\boxed{n \text{ distributed web caches}}$

Recal hash function:-

- Universal.
  We are given a set $\mathcal{H}$ of hash functions mapping from $\mathcal{U} \rightarrow \{0, 1, 2 \cdots m-1\}$.

  such that $\forall\, x, y \in \mathcal{U}$, where $x \neq y$.

$$\left| \{ h_a \in \mathcal{H} \mid h_a(x) = h_a(y) \} \right| = \frac{|\mathcal{H}|}{m}$$

i.e.; the probability that x & y Collide is $\frac{1}{m}$, if we choose h_a randomly from $\mathcal{H}$.

- 2-wise Independent:

$$\mathcal{H} = \{ h_{a,b} \mid a \in \{1, 2 \ldots p-1\} \text{ \& } b \in \{0, 1, 2 \ldots p-1\} \}.$$

  where

$$h_{a,b}(x) = (ax + b \bmod p) \bmod n$$

→ Using a 2-wise independent family of hash functions, we can create a perfect hashing.

→ Perfect hashing only works well if the number of machines does not change during the process.

→ If the number of machines changes:

① Change the n in $h_{a,b}$ to n' to get $h'_{a,b}$

→ By doing so, we need move almost all item to their new location.

② Keep n unchanged and thus no moving

→ The new machine is not used. will create load imbalance.

→ we need a strategy that does not incur too many re-hashing and in the mean time keep the load of all machines almost balanced.

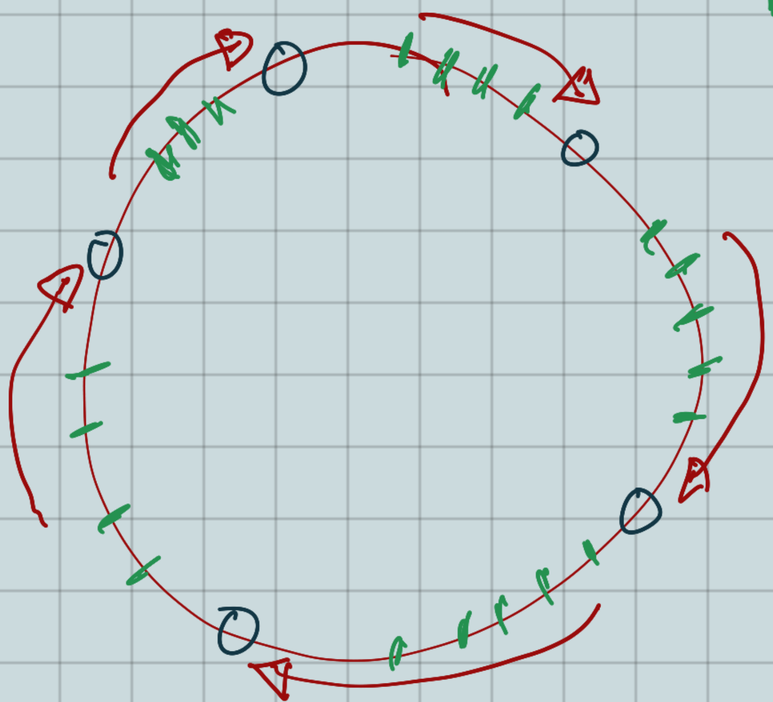
Basic idea:

→ Each machine is mapped to a random real numbers in the interval [0, 1]

→ Each item is mapped to a random real number in the interval [0, 1]

→ Store each item in first machine on its right.
If no cache ~~on~~ the right, then store the item in the cache with the smallest number.

o → machine.
| → items.



Implementation:-

- To dynamically maintain machines & items, we need to maintain a Binary Search Tree (BST), whose keys are the values assigned to the machines.
- Let h_i & h_m be respectively the functions that we used to hash items and machines to the interval [0, 1]

- To insert an item x:
  - Find successor of $h_i(x)$ in the ST
  - If no successor then find the mallest h_m value.
  - Store x in the retorned m ine.

- To delete an item x:
  - Find the successor of $h_i(x)$ in the BST.
  - If no successor th find the smallest h_m value
  - Delete x in the urned machine

→ To insert a new machine Y

- There may be some existing items that should be stored in the new machine Y, but these items are all stored in the successor of $h_m(Y)$

- Find the successor of $h_m(Y)$ in the BST
- Move all items whose h_i value is less than $h_m(Y)$ to the newly inserted machine Y.

→ To delete an existing machine Y:
- Find the successor of $h_m(Y)$ in the BST
- Move all items in Y to the returned machine

Bounds:

Lemma 1: With high probability, no machine owns more than

$$O\left(\frac{\lg n}{n}\right).$$

Lemma 2: With high probability, the size of the smallest interval assigned to a machine is

$$O\left(\frac{1}{n^2}\right)$$

Proof:- Fix some interval I of length $\frac{2\lg n}{n}$.

Pr [no machine lands in I]

$$\left(1-\frac{2\lg n}{n}\right)^n = \left(\left(1-\frac{2\lg n}{n}\right)^{\frac{n}{2\lg n}}\right)^{2\lg n} \approx \frac{1}{n^2}$$

Equally split $[0,1]$ to $\frac{n}{2\lg n}$ such intervals.

By union bound,

Pr [every one of these intervals contains at least 1 machine]

$$1-\frac{n}{2\lg n}\cdot\frac{1}{n^2} > 1-\frac{1}{n}$$

With high Pr. each machine owns an interval of length at most $\frac{4\lg n}{n}$.

Lemma 3: When a machine is added, the expected number of items that move to the newly added machine is $\frac{m}{n+1}$.

Random Trees:

- The actual motivation of random trees is to relieve the hot spots on the web.
- If only a root server is handling all the request, it will quickly become the bottleneck.
- The goal is to use a set of proxy caches and requests can be distributed among them.

Implementation:-

- choose a d-ary tree with n virtual nodes V.
- The root server is located in the root of this d-ary tree.
- We choose a consistent hash ftn.

$h: V \to C$

→ For each request of a page,

→ choose a random leaf v in the random d-ary tree
→ Ask the virtual nodes in the $v \to r$ path one by one.
- For each node U in the path, if the cache $h(U)$ contains the requested page, then return the page
- otherwise, go to the parent of U on the $v \to r$ path and increment page's counter on the local cache. (i.e $h(U)$ that missed the request)

→ for any local cache, if the counter for a page reaches a fixed threshold denoted by q, then the cache stores the page.

→ At the begining all pages are only on the root server. As it goes and by receiving the requests, the popular pages will spread downward in the tree.
→ This guarantees that no local cache gets too many requests for any page.

Lemma 4: Each cache that is not mapped to a leaf on the random tree is asked for the same page at most

$$O\left(dq \frac{\lg n}{\lg \lg n}\right) \text{ time w.h.p}$$