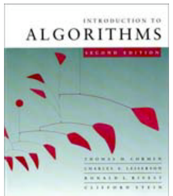
Consistent Hashing

PPT by Brandon Fain

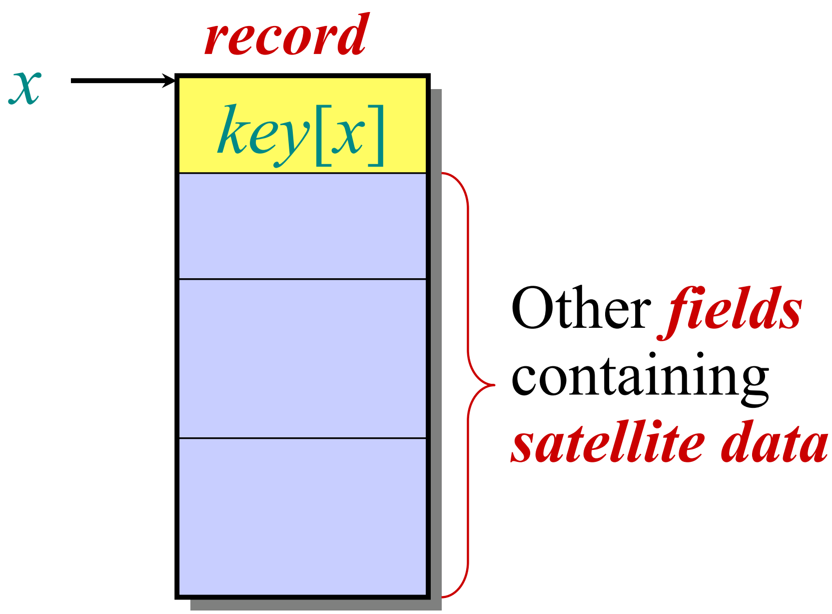
Outline

- Review Hashing
- Motivation: Caching Webpages
- Consistent Hashing



Symbol-table problem

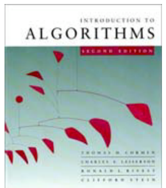
Symbol table S holding n ***records***:



Operations on S:

- INSERT(S, x)
- DELETE(S, x)
- SEARCH(S, k)

How should the data structure S be organized?



Direct-access table

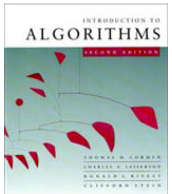
IDEA: Suppose that the keys are drawn from the set $U \subseteq \{0, 1, \ldots, m-1\}$, and keys are distinct. Set up an array $T[0 \, .. \, m-1]$:

$$T[k] = \begin{cases} x & \text{if } x \in K \text{ and } key[x] = k, \\ \text{NIL} & \text{otherwise.} \end{cases}$$

Then, operations take $\Theta(1)$ time.

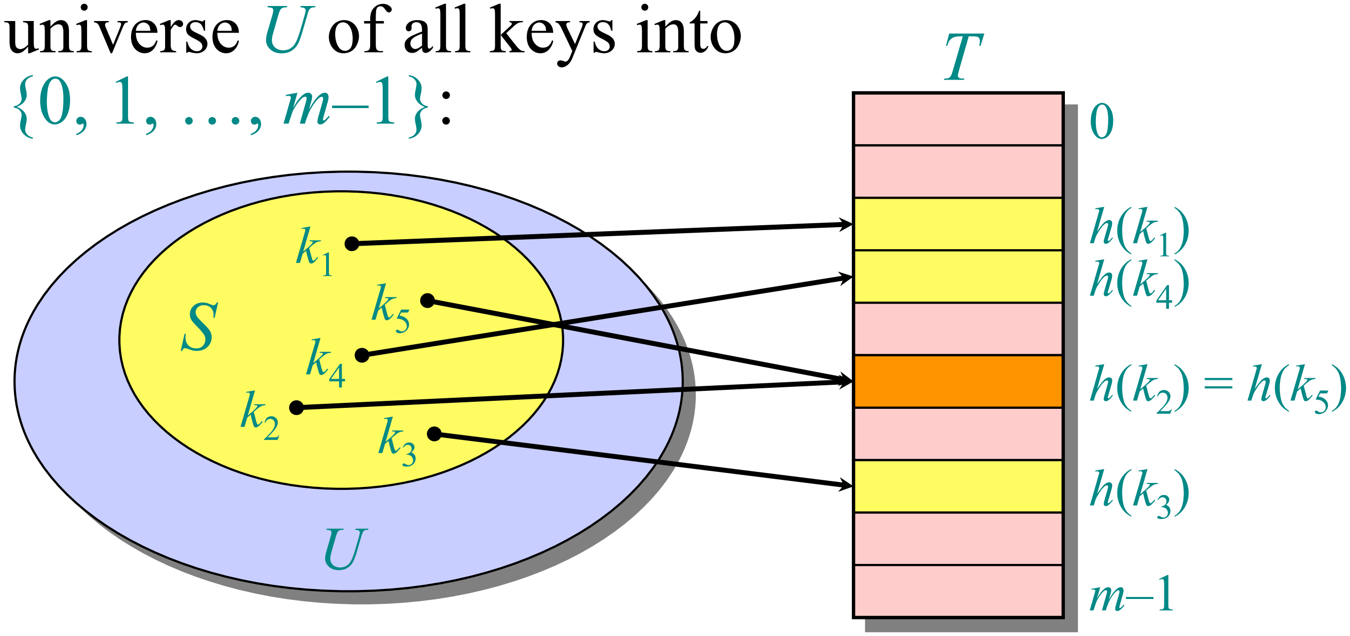
Problem: The range of keys can be large:

- 64-bit numbers (which represent 18,446,744,073,709,551,616 different keys),
- character strings (even larger!).

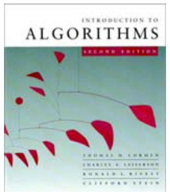


Hash functions

Solution: Use a ***hash function*** h to map the universe U of all keys into $\{0, 1, \ldots, m-1\}$:



When a record to be inserted maps to an already occupied slot in T, a ***collision*** occurs.



Average-case analysis of chaining

We make the assumption of ***simple uniform hashing:***

- Each key $k \in S$ is equally likely to be hashed to any slot of table T, independent of where other keys are hashed.

Let n be the number of keys in the table, and let m be the number of slots.

Define the ***load factor*** of T to be

$$\alpha = n/m$$

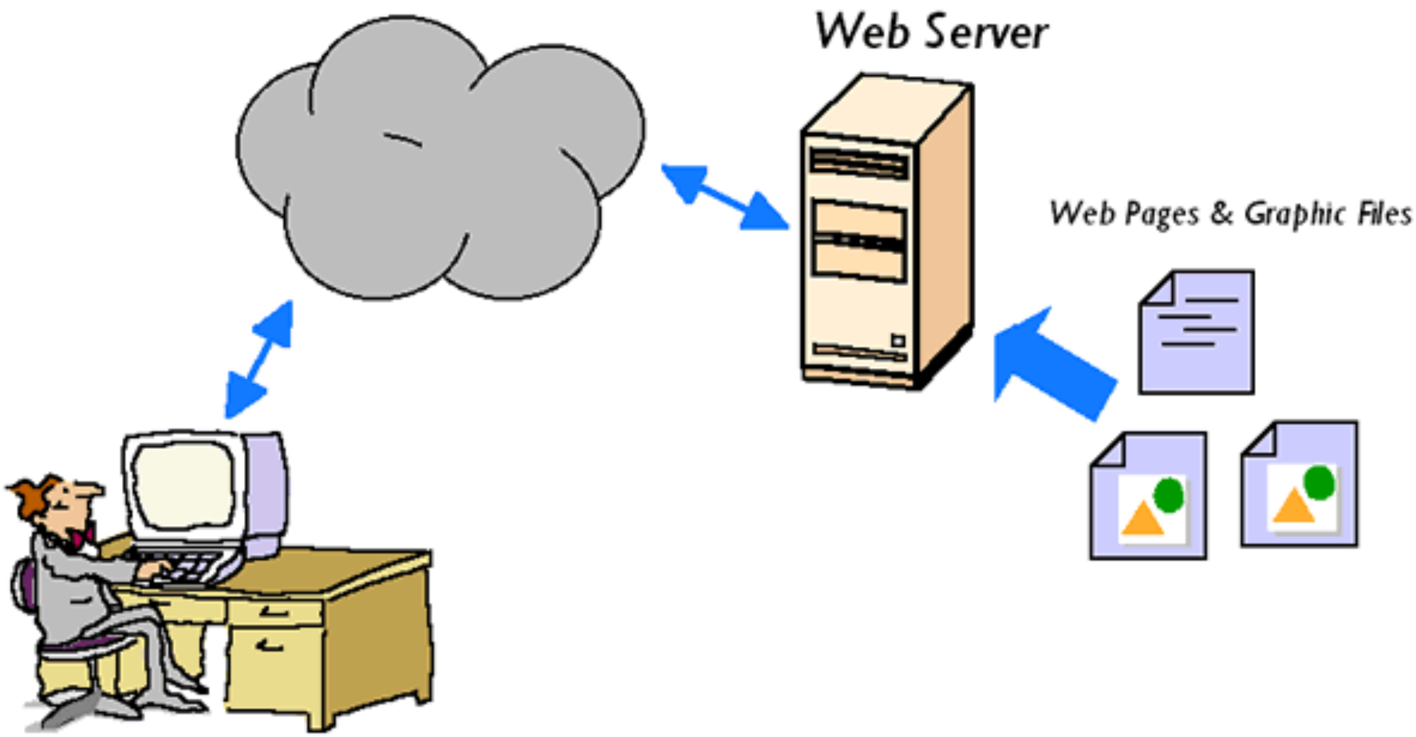
$= \text{average number of keys per slot.}$

Outline

- ~~Review Hashing~~
- Motivation: Caching Webpages
- Consistent Hashing

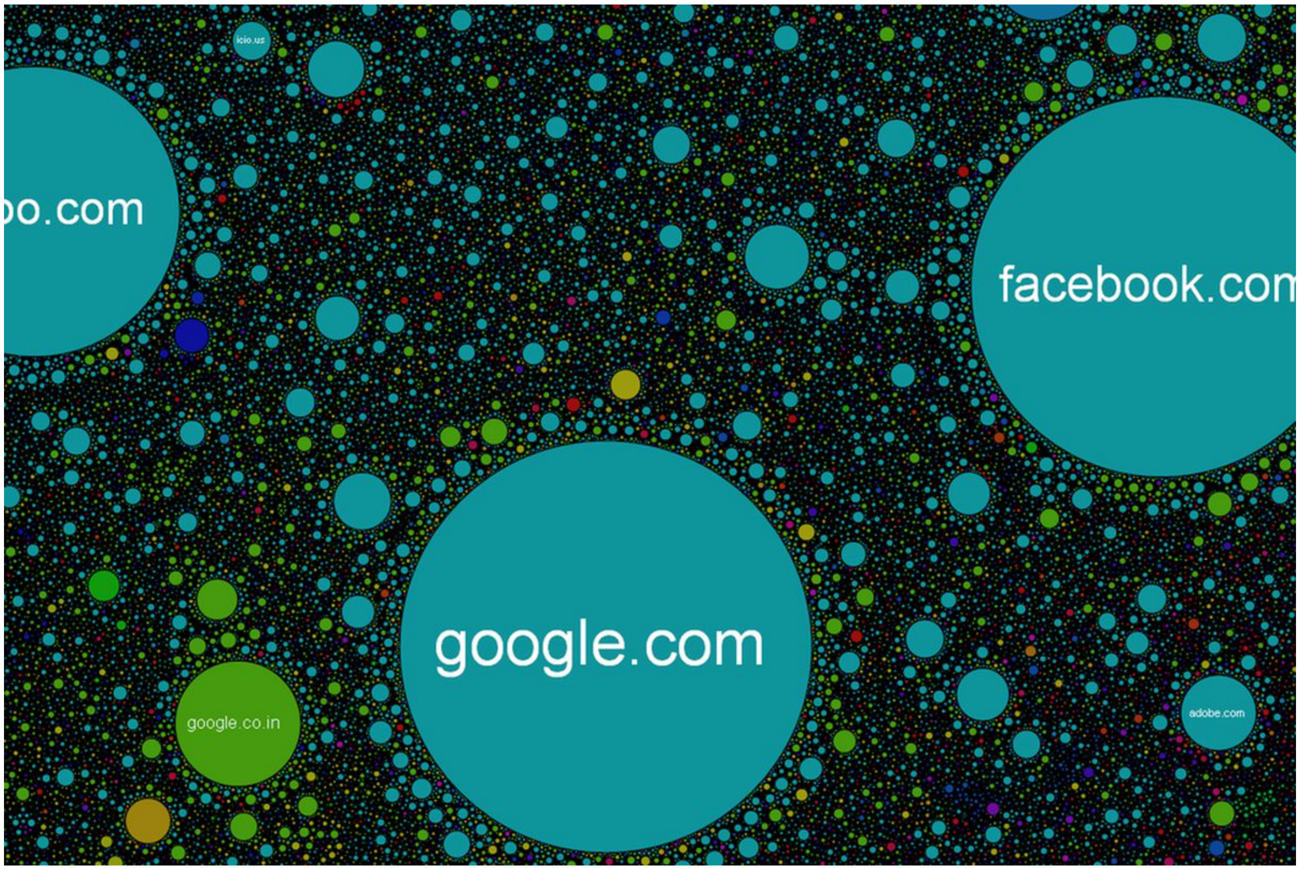
Caching Webpages

- The usual model:



Caching Webpages

- Reality:

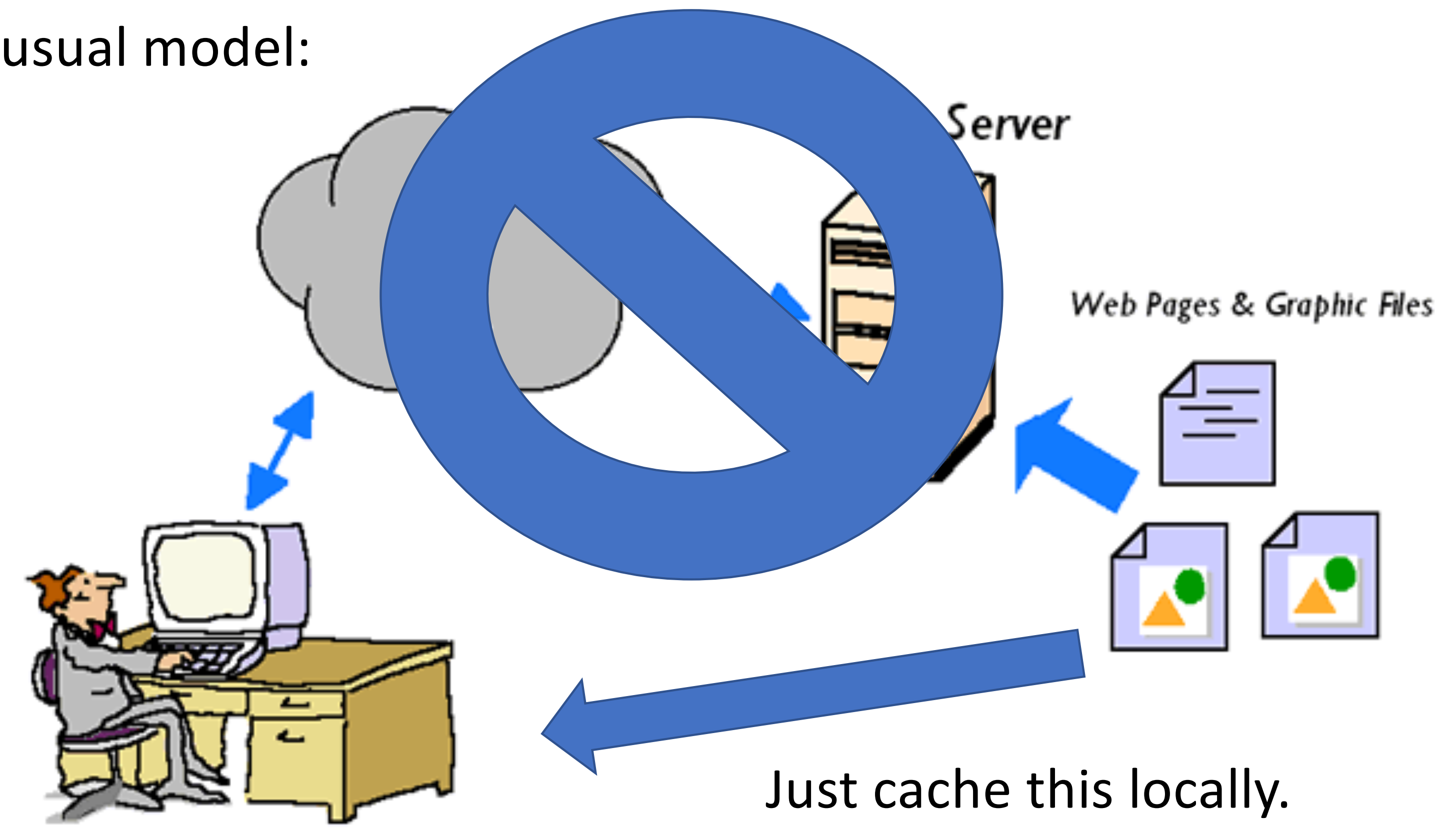


Caching Webpages

Can we cut out the server bottleneck?

Caching Webpages

- The usual model:



Caching Webpages – Advantages

- Users get much faster response times from webpages.
- Overall network congestion is decreased.
- Server load is decreased.
- It’s a win win!
  - Well…except that it costs space. Maybe too much for one device.

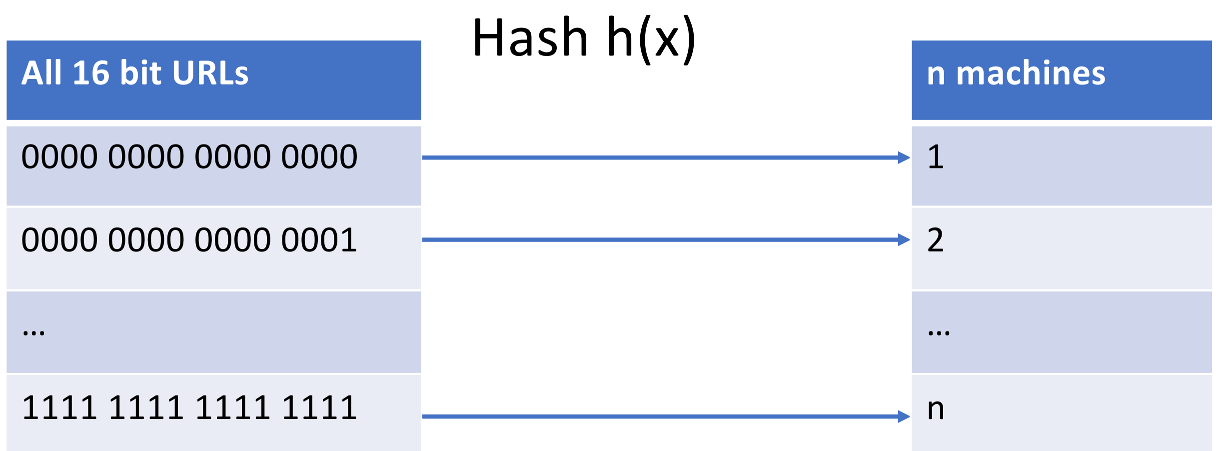
Caching Webpages

- Better yet, couldn't *multiple* users/devices share a common cache of recent urls?

- **Problem:** Who stores what? When we try to visit google.com, how do we know which device in our local network has the page in cache?

- **Solution:** Hashing!

Caching Webpages



Set h(x) to be something like MD5(x) mod n

(MD5(x) is a widely used hash function producing a 128-bit hash of x)

The expected load on any machine will just be m/n, if there are m webpages cached.

Caching Webpages

- **Problem:** What happens if we add or take away a device from this caching scheme?

- We could just set h(x) to be something like MD5(x) mod (n+1).

- But then we have to move almost all m cached pages between devices.

- For a problem at this scale on the internet, devices can come and go too often for this to be even remotely feasible.

Outline

- ~~Review Hashing~~
- ~~Motivation: Caching Webpages~~
- Consistent Hashing

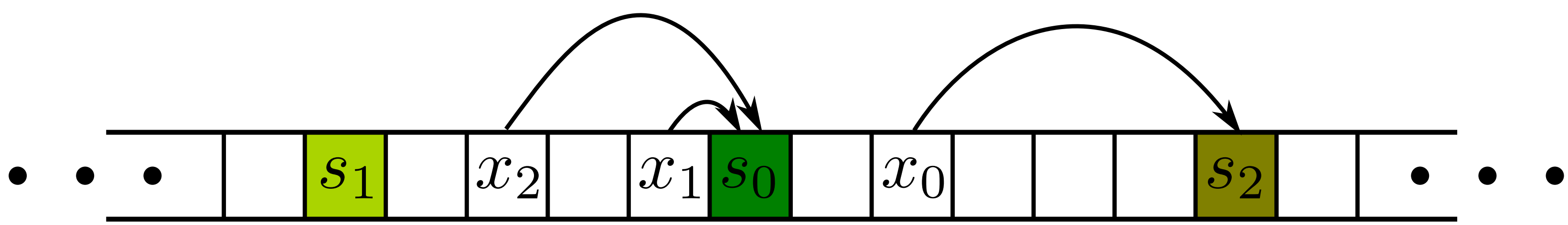
Consistent Hashing

- We want a way to increase or decrease the number of “buckets” in our hash table *without* needing to shuffle a lot of data.

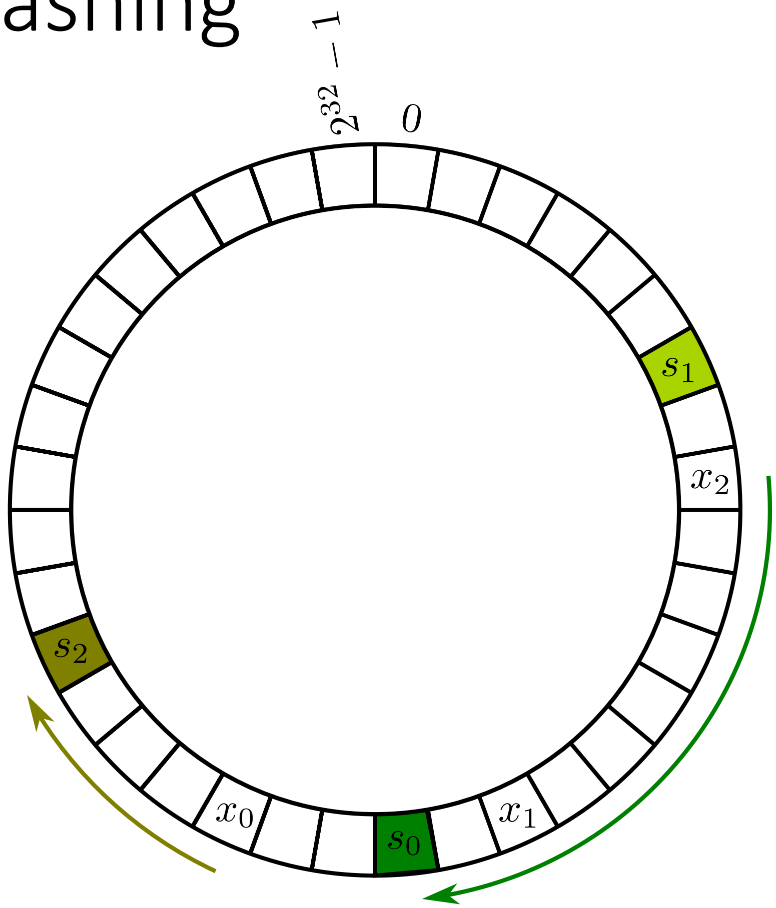
- **Key idea:** Don’t hash to machines directly. Hash to values, and *also* hash the names of the machines.

- To lookup a page, find the active machine whose hash value is closest (to the right) to the hash value of the page.

Consistent Hashing



Consistent Hashing



Consistent Hashing

- With m webpages and n machines, we still have expected load m/n per machine.
- Now we only have to move m/n pages, in expectation, when we add or remove a machine.
- Note that we can even do this in lazy execution!

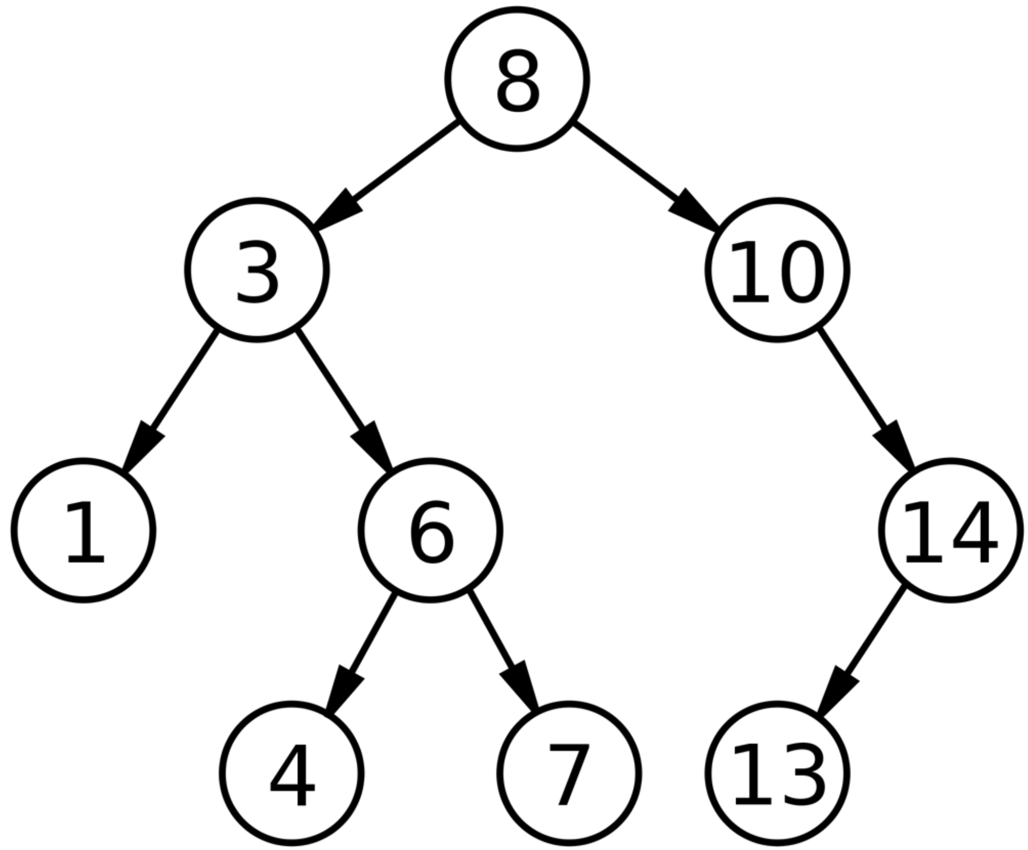
Consistent Hashing

- **Problem.** How do we actually implement the “find the active machine whose hash value is closest (to the right) to the hash value of the page” idea?

- **Solution:**
- Maintain a binary search tree on the machines, sorted by hash values.
- Then given the hash value of a page, we can find it’s machine in O(log(n)) time, assuming the tree is balanced.
  - Note – one should use a red and black tree, as the tree will be changing frequently and needs to stay balanced.

Consistent Hashing

For example, suppose you want to know to what machine you should cache a page hashed to 5.



Consistent Hashing

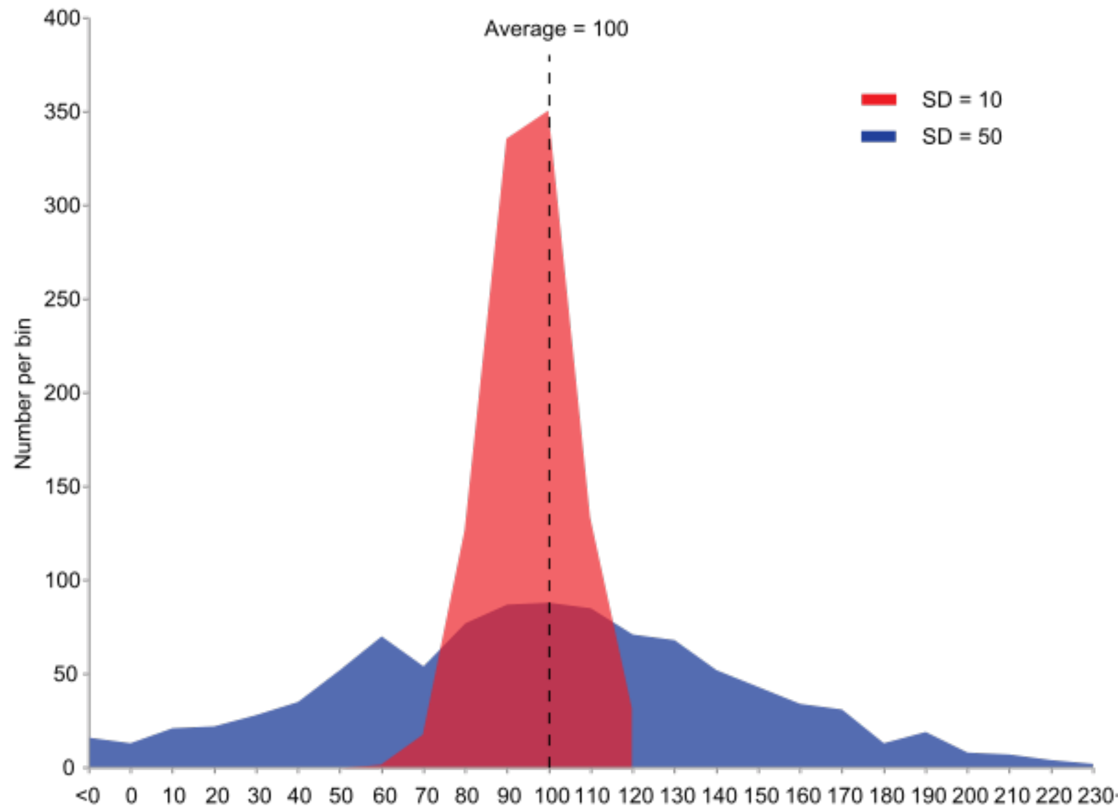
- Unfortunately, we have introduced another problem.

- In *expectation*, the load per machine should still be m/n. But expectations aren't everything…

Consistent Hashing

- These two distributions have the same expectation…

- But different **variance**. Recall that the variance of a random variable X is $\mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$.



Consistent Hashing

- Let *X* be the random variable for the load on a machine after caching *m* pages on *n* machines using our consistent hashing scheme.

- **Problem.** *X* has substantially higher variance than one would typically expect in hashing applications. Why?

Consistent Hashing

- The standard idea is to create multiple *logical* machines for each physical machine. Everything is as before, except multiple logical machines actually get stored on the same device.

- Another general purpose idea for reducing variance in hashing is to use multiple hash functions.

Consistent Hashing - Akamai

Market Capitalization = 12.3 billion USD

Consistent Hashing - Akamai