Linear Probing

Outline for Today

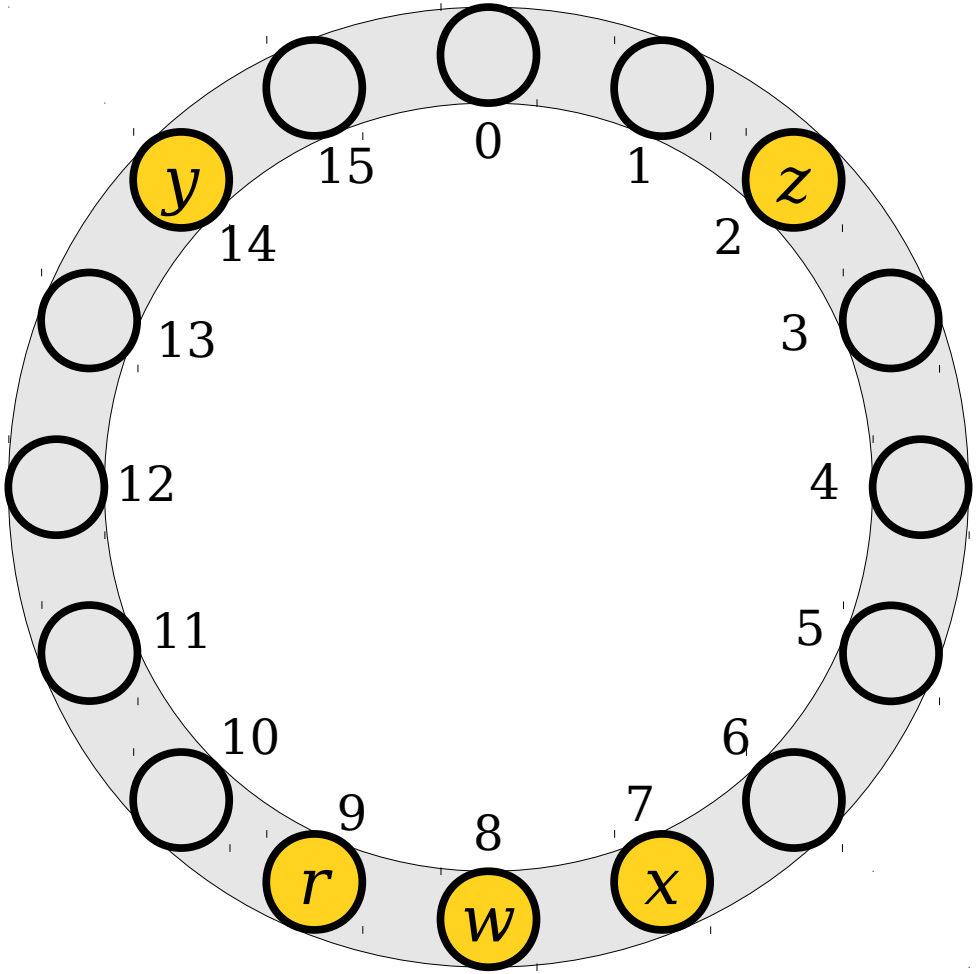
- ***Linear Probing Hashing***
  - A simple and lightning fast hash table implementation.
- ***Analyzing Linear Probing***
  - Why the degree of independence matters.
- ***Fourth Moment Bounds***
  - Another approach for estimating frequencies.

Hashing Strategies

- All hash table implementations need to address what happens when collisions occur.
- Common strategies:
  - ***Closed addressing:*** Store all elements with hash collisions in a secondary data structure (linked list, BST, etc.)
  - ***Perfect hashing:*** Choose hash functions to ensure that collisions don't happen, and rehash or move elements when they do.
  - ***Open addressing:*** Allow elements to “leak out” from their preferred position and spill over into other positions.
- Linear probing is an example of open addressing.
- We'll see a type of perfect hashing (***cuckoo hashing***) on Thursday.

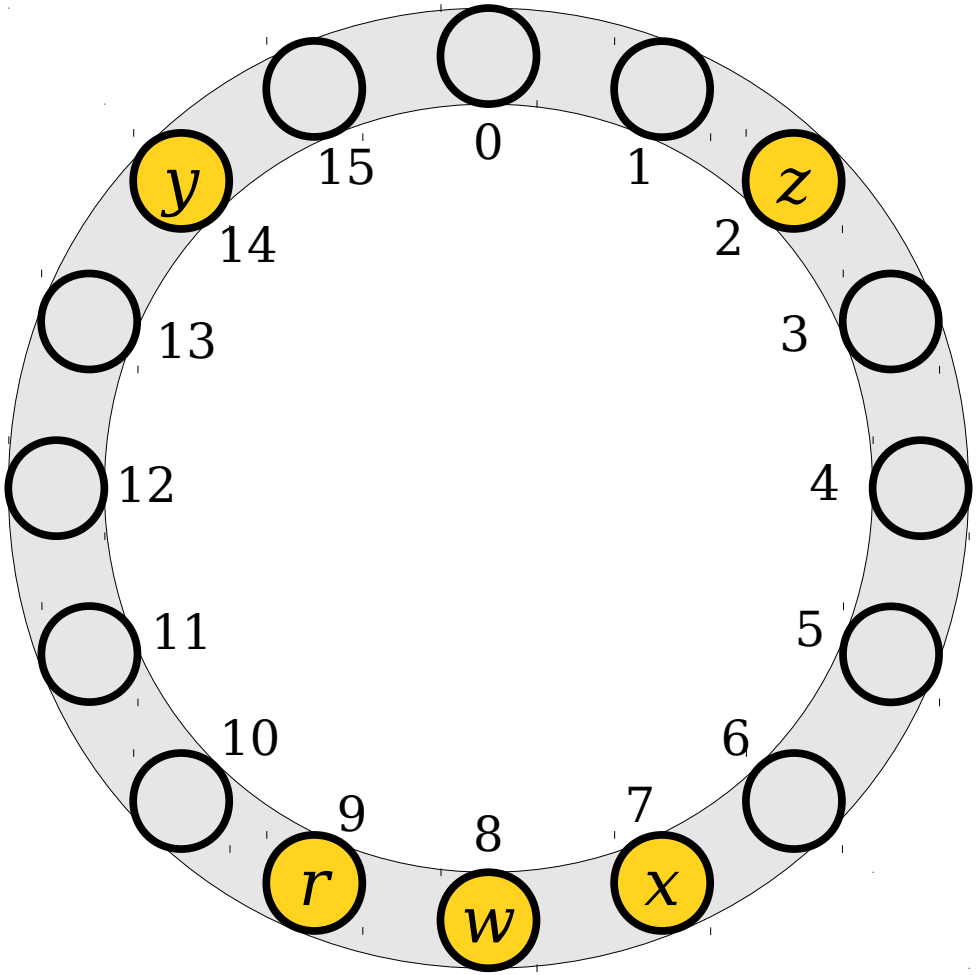
Linear Probing

- ***Linear probing*** is a simple open-addressing hashing strategy.
- To insert an element x, compute $h(x)$ and try to place x there.
- If it's full, keep moving through the array, wrapping around at the end, until a free spot is found.



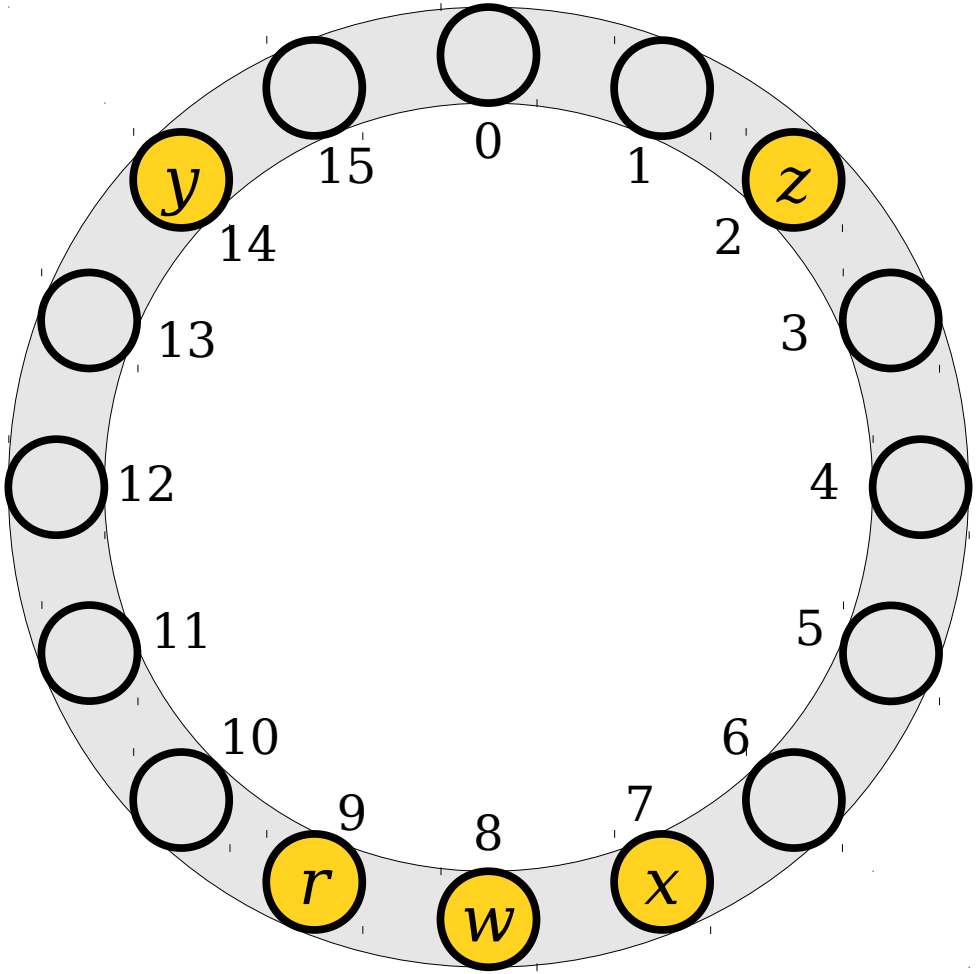
Linear Probing

- To look up an element x, compute $h(x)$ and start looking there.
- Move around the ring until either the element is found or a blank spot is detected.
- (We'll assume the load factor prohibits us from inserting so many elements that there are no free spaces.)



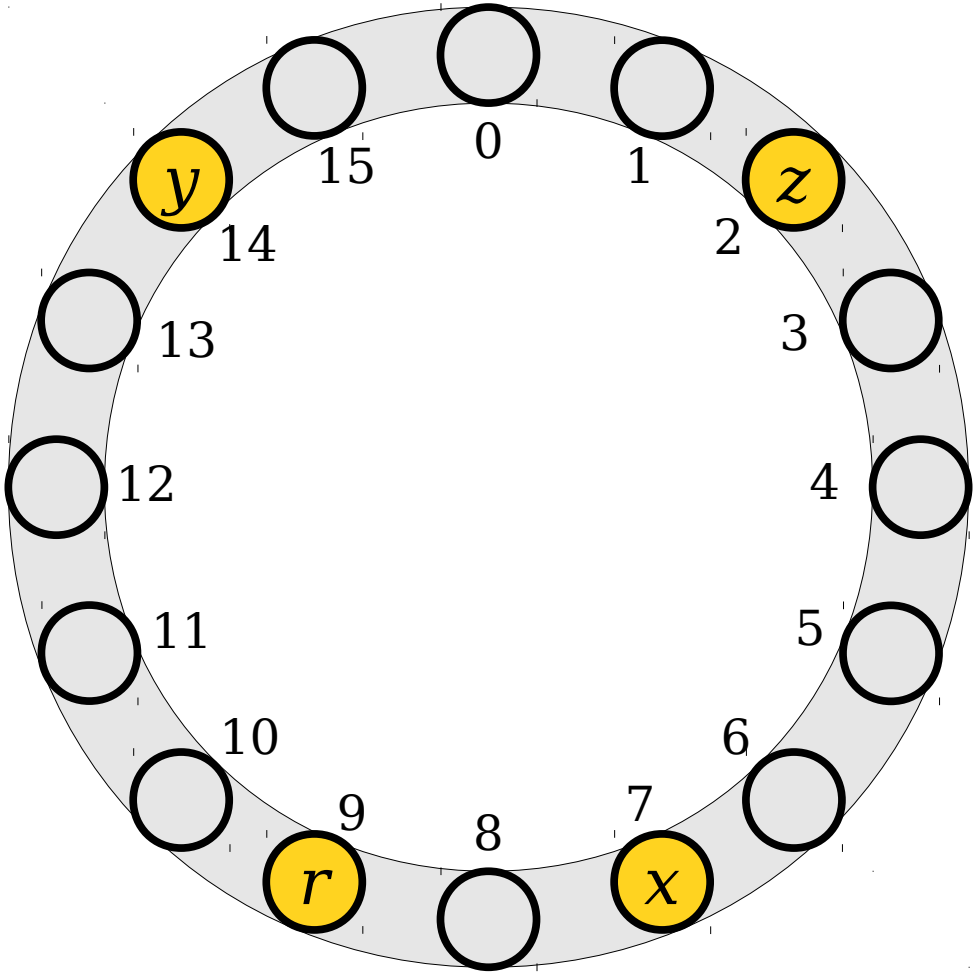
Linear Probing

- Deletions are a bit trickier than in chained hashing.
- We cannot just do a search and remove the element where we find it.
- Why?



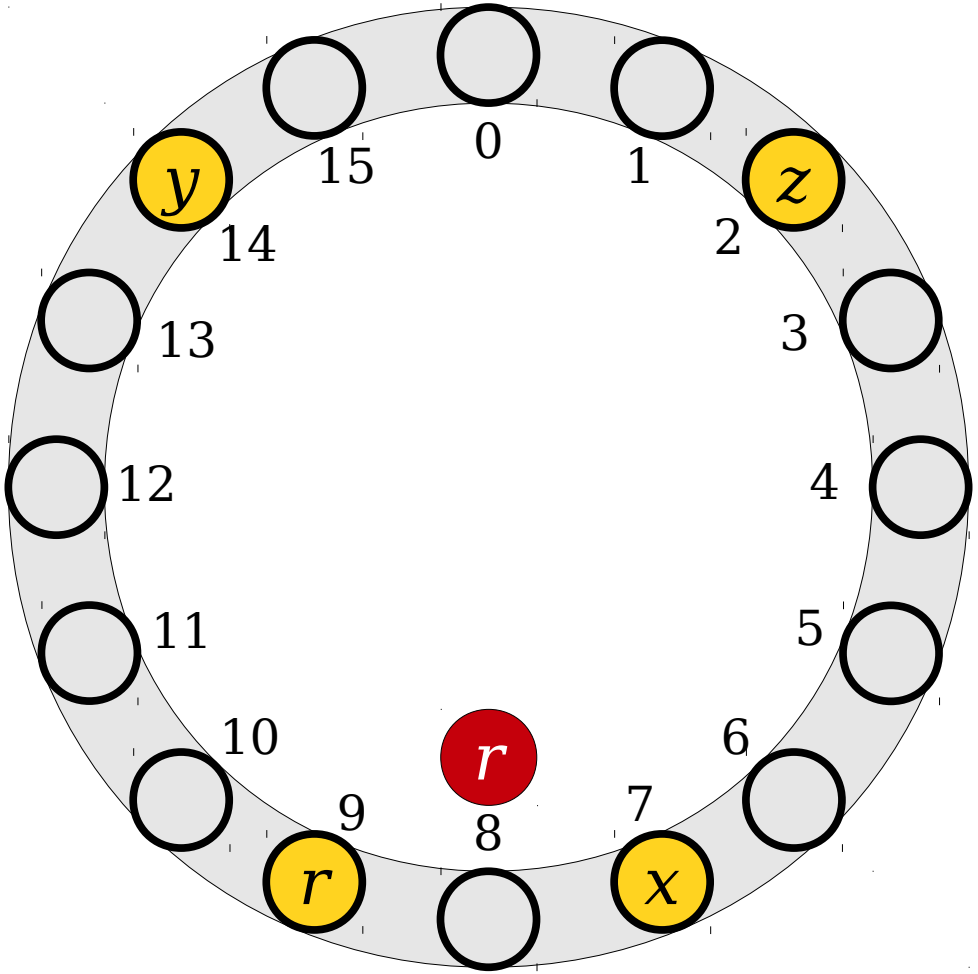
Linear Probing

- Deletions are a bit trickier than in chained hashing.
- We cannot just do a search and remove the element where we find it.
- Why?



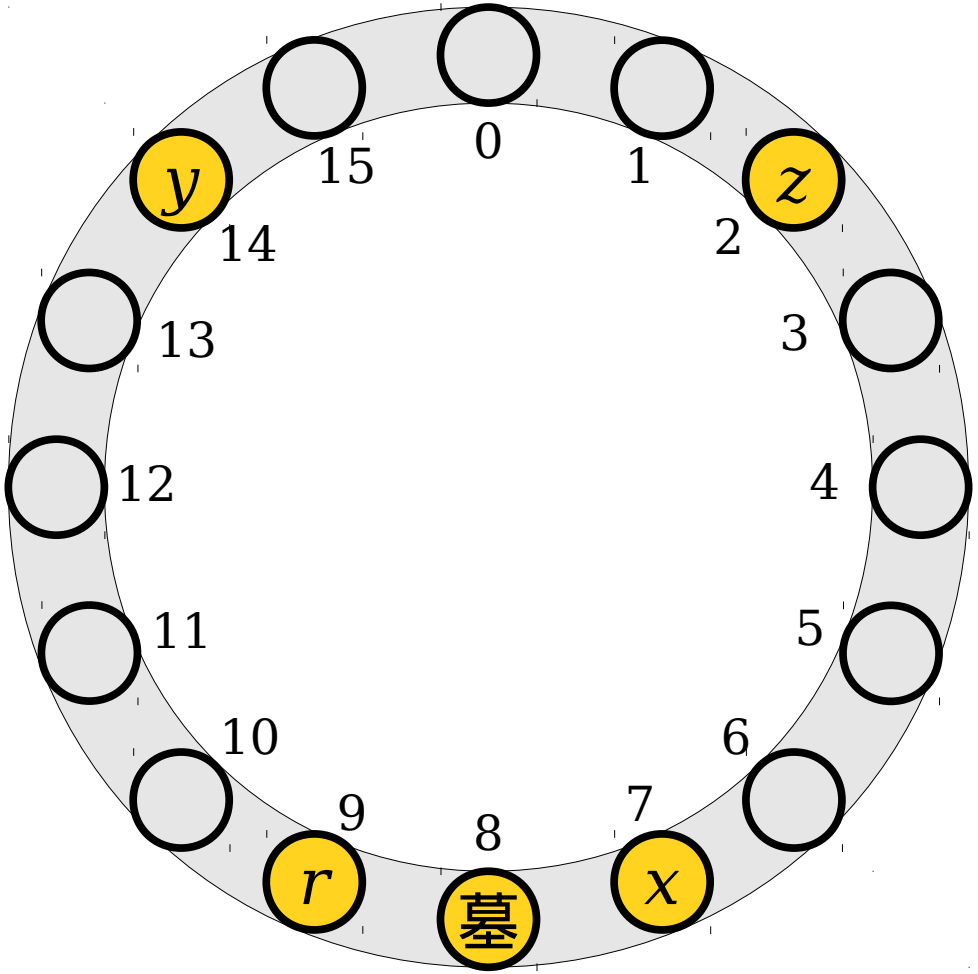
Linear Probing

- Deletions are a bit trickier than in chained hashing.
- We cannot just do a search and remove the element where we find it.
- Why?



Linear Probing

- Deletions are often implemented using ***tombstones***.
- When removing an element, mark that the cell is empty and was previously occupied.
- When doing a lookup, don't stop at a tombstone. Instead, keep the search going.
  - You need to watch out for wraparounds.
  - When inserting, feel free to replace any tombstone you encounter.

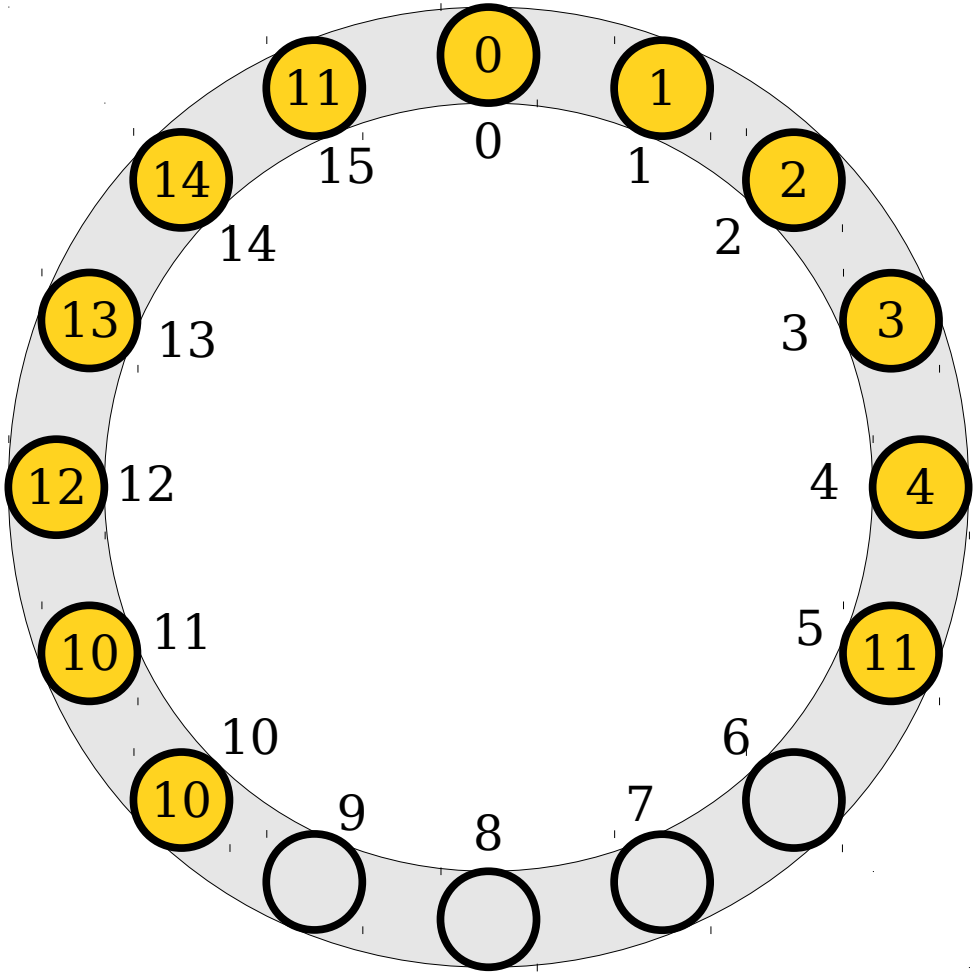


Linear Probing in Practice

- In practice, linear probing is one of the fastest general-purpose hashing strategies available.
- This is surprising – it was originally invented in 1954! It's pretty amazing that it still holds up so well.
- Why is this?
  - ***Low memory overhead:*** just need an array and a hash function.
  - ***Excellent locality:*** when collisions occur, we only search in adjacent locations in the array.
  - ***Great cache performance:*** a combination of the above two factors.

The Weakness

- Linear probing exhibits severe performance degradations when the load factor gets high.
- The number of collisions tends to grow as a function of the number of existing collisions.
- This is called ***primary clustering***.



Time-Out for Announcements!

Project Proposals

- Project proposals were due today at 3:00PM.
  - Didn't submit yet? Please do so ASAP – if not, you'll end up on a random team with a random topic.
- We'll run a matchmaking algorithm to assign topics and aim to get back to everyone by Thursday.
- We're really excited to see what you all come up with!

Problem Set Four

- Problem Set Four is due on Thursday at 3:00PM.
- Need help? Ask on Piazza or stop by office hours!

Back to CS166!

Analyzing Linear Probing

Some Brief History

- The first rigorous analysis of linear probing was done by Don Knuth in 1962. You can read it on the course website.
- Knuth's analysis assumed that the underlying hash function was a truly random function.
- Under this assumption, the expected cost of a successful lookup is $O(1 + (1 - \alpha)^{-1})$, where α is the load factor, and the expected cost of an insertion or unsuccessful lookup is $O(1 + (1 - \alpha)^{-2})$.
  - If we have n elements and m buckets, then $\alpha = n / m$.
- Notice that this is $O(1)$ for any fixed α, but as α grows, this runtime gets progressively worse, as expected.

Hash Function Strength

- The preceding analysis assumes that the hash functions used are truly random functions, which is too strong an assumption in practice.
- Typically, the "strength" of hash functions in data structures is measured by their independence.
- A family of hash functions $\mathscr{H}$ from a universe $\mathscr{U}$ to the set $[m]$ is called ***k-independent*** if
  - for any $x \in \mathscr{U}$, and for any h drawn uniformly from $\mathscr{H}$, the random variable $h(x)$ is uniformly-distributed over $[m]$; and
  - for any distinct $x_1, \ldots, x_k \in \mathscr{U}$ and any h drawn uniformly from $\mathscr{H}$, the random variables $h(x_1), \ldots, h(x_k)$ are independent.

Comparison: Chained Hashing

- ***Theorem:*** The expected cost of a lookup in chained hashing with 2-independent hash functions is O(1 + α).
- ***Proof:*** Consider any key x_i and let X_{ij} be an indicator variable that's 1 if there's a collision between x_i and x_j and 0 otherwise. Then

$$\begin{aligned} E[\sum_{i \neq j} X_{ij}] &= \sum_{i \neq j} E[X_{ij}] \\ &= \sum_{i \neq j} \Pr[h(x_i) = h(x_j)] \\ &= \sum_{i \neq j} \sum_{k=1}^{m} \Pr[h(x_j) = k \mid h(x_i) = k] \Pr[h(x_i) = k] \end{aligned}$$

This term is $1/m$ because a 2-independent hash function distributes each hash code uniformly over the buckets.

Comparison: Chained Hashing

- ***Theorem:*** The expected cost of a lookup in chained hashing with 2-independent hash functions is O(1 + α).
- ***Proof:*** Consider any key x_i and let X_{ij} be an indicator variable that's 1 if there's a collision between x_i and x_j and 0 otherwise. Then

$$\begin{aligned} E[\sum_{i \neq j} X_{ij}] &= \sum_{i \neq j} E[X_{ij}] \\ &= \sum_{i \neq j} \Pr[h(x_i) = h(x_j)] \\ &= \sum_{i \neq j} \sum_{k=1}^{m} \Pr[h(x_j) = k \mid h(x_i) = k] \Pr[h(x_i) = k] \end{aligned}$$

This term is $1/m$ because, conditioning on knowing $h(x_i)$, the hash code of any other key is independent and uniformly-distributed.

Comparison: Chained Hashing

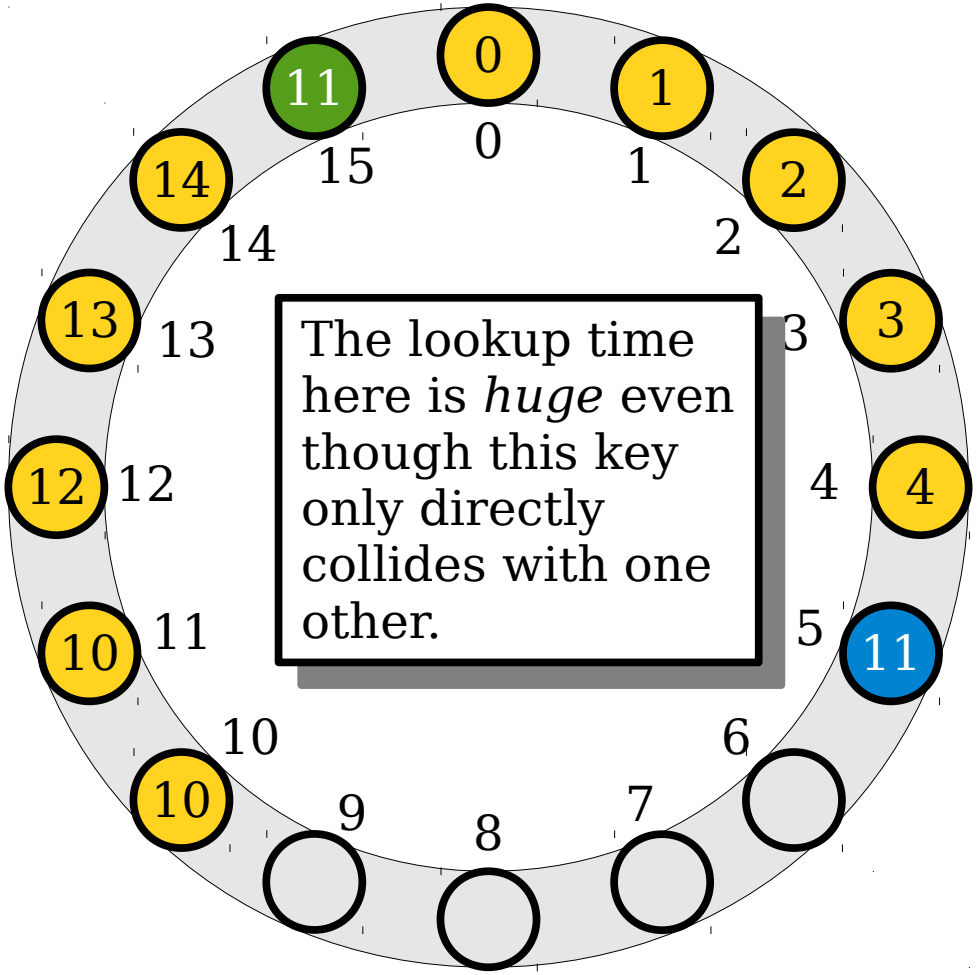
- ***Theorem:*** The expected cost of a lookup in chained hashing with 2-independent hash functions is O(1 + α).
- ***Proof:*** Consider any key x_i and let X_{ij} be an indicator variable that's 1 if there's a collision between x_i and x_j and 0 otherwise. Then

$$\begin{aligned}
E[\sum_{i \neq j} X_{ij}] &= \sum_{i \neq j} E[X_{ij}] \\
&= \sum_{i \neq j} \Pr[h(x_i) = h(x_j)] \\
&= \sum_{i \neq j} \sum_{k=1}^{m} \Pr[h(x_j) = k \mid h(x_i) = k] \Pr[h(x_i) = k] \\
&= \sum_{i \neq j} \sum_{k=1}^{m} \frac{1}{m^2} \\
&= \sum_{i \neq j} \frac{1}{m} \\
&\leq \alpha
\end{aligned}$$

The cost of a lookup is at most O(1) plus the number of collisions, hence the expected lookup time is O(1 + α). ■

Why Linear Probing is Different

- In chained hashing, collisions only occur when two values have exactly the same hash code.
- In linear probing, collisions can occur between elements with entirely different hash codes.
- To analyze linear probing, we need to know more than just how many elements collide with us.



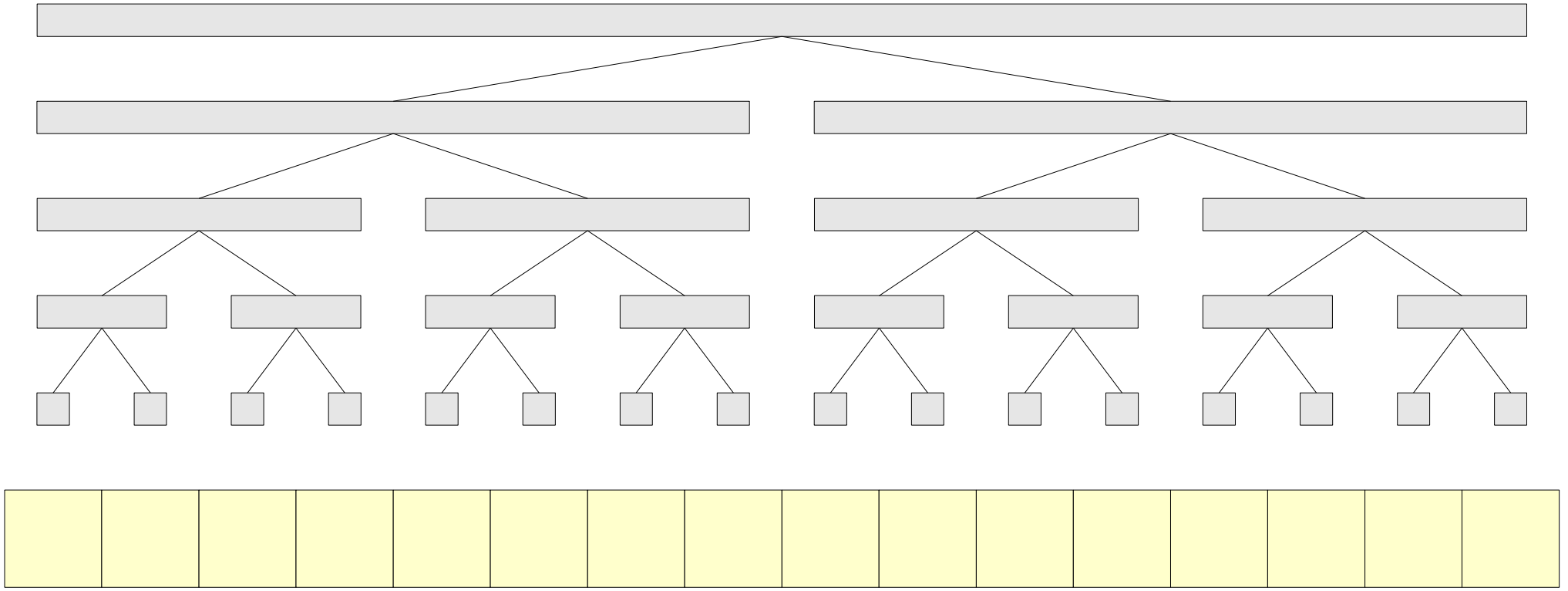
Question: What impact – if any – does the degree of independence of our hash functions have in linear probing?

Analyzing Linear Probing

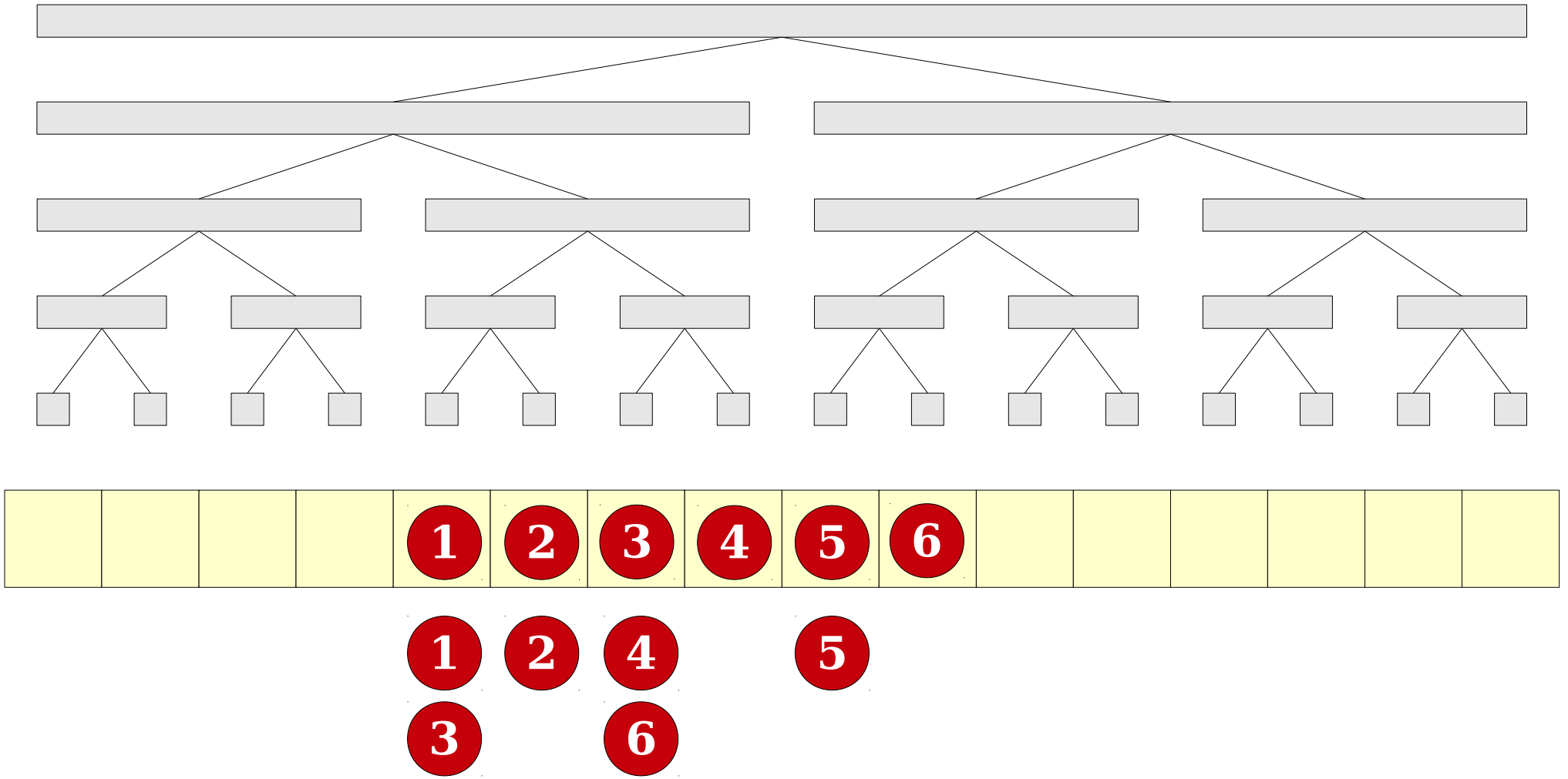
- When looking at k-independent hash functions, the analysis of linear probing gets significantly more complex.
- Where we're going:
  - ***Theorem:*** Using 2-independent hash functions, we can prove an $O(n^{1/2})$ expected cost of lookups with linear probing, and there's a matching adversarial lower bound.
  - ***Theorem:*** Using 3-independent hash functions, we can prove an $O(\log n)$ expected cost of lookups with linear probing, and there's a matching adversarial lower bound.
  - ***Theorem:*** Using 5-independent hash functions, we can prove an $O(1)$ expected cost of lookups with linear probing.
- These results may seem completely counterintuitive now, but they make a lot of sense when we dive into the math. In fact, they hit at a key idea in the design and analysis of randomized data structures.

The Setup

This data structure is called a ***segment tree***. Look it up – it's interesting!



The Setup

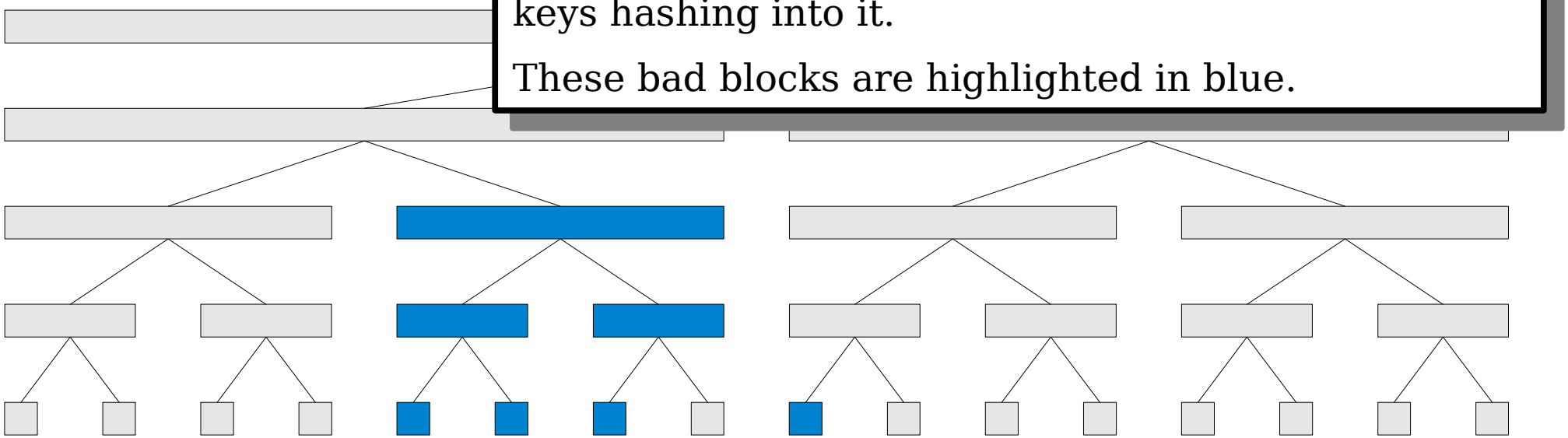


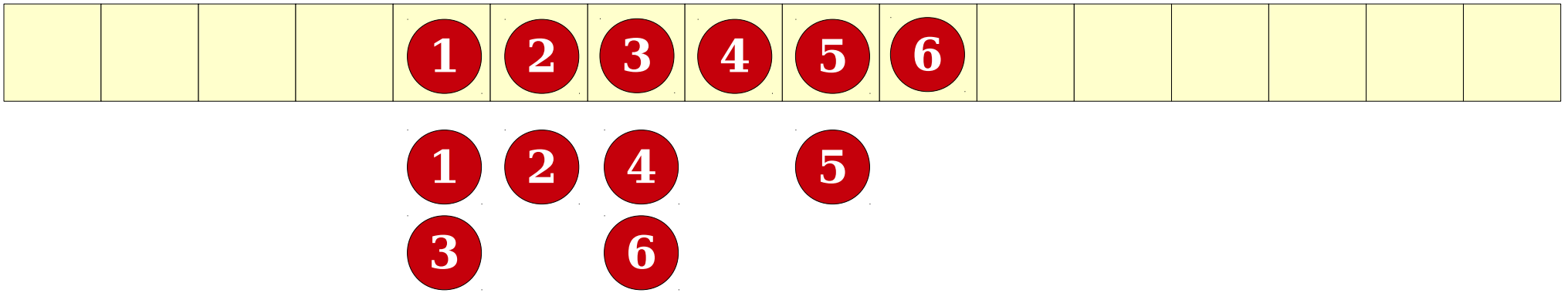
The Setup

Let's use ⅓ as a load factor. On expectation, a block of size 2^l will have $\frac{1}{3} \cdot 2^l$ keys hash into it.

We'll say that a ***bad block*** is one with at least $\frac{2}{3} \cdot 2^l$ keys hashing into it.

These bad blocks are highlighted in blue.





A Key Theorem

- ***Theorem:*** The probability that a key x_i is in a run of length between 2^l and 2^{l+1} is at most

  $b \cdot \Pr[$ the block of length 2^l above $h(x_i)$ in the tree is bad $]$

  for some fixed universal constant b.
- ***Proof idea:*** Use the pigeonhole principle to show that if a certain window of blocks of length 2^l in the segment tree aren't bad, then there aren't enough keys to make a run of length 2^l. Then, apply the union bound.
  - See Thorup's lecture notes for details.
- We will use this theorem to determine the expected cost of a lookup in a linear probing hash table.

Analyzing the Runtime

- The cost of looking up some key x_q is bounded from above by the length of the run containing x_q.
- The expected cost of performing a lookup is therefore at most

$$O(1) \cdot \sum_{l=0}^{O(\log n)} 2^l \Pr[x_q \text{ is in a run of length } 2^l]$$

- Our previous theorem tells us that this cost is

$$O(1) \cdot \sum_{l=0}^{O(\log n)} 2^l \Pr[\text{the } 2^l \text{ block above } h(x_q) \text{ is bad}]$$

- If we can determine the probability that a given block is bad, then we'll have a bound on the expected lookup cost for x_q.

Bad Blocks

- ***Recall:*** On expectation, a block of size 2^l will have $\frac{1}{3} \cdot 2^l$ elements in it.
- A bad block is one that has at least $\frac{2}{3} \cdot 2^l$ elements in it.
- Pick a fixed size 2^l. Let the random variable X represent the number of keys that hash into the block of that size near $h(x_q)$. We're then interested in

$$\Pr[\, X \geq \tfrac{2}{3} \cdot 2^l \,].$$

- Let's define $\mu = \mathrm{E}[X] = \frac{1}{3} \cdot 2^l$. Then the above quantity is equivalent to

$$\Pr[\, X \geq 2\mu \,],$$

or, equivalently,

$$\Pr[\, X - \mu \geq \mu \,].$$

A note: Everything in this analysis implicitly is conditioned on knowing on where x_q hashed to. For simplicity we'll omit the notation for it, but keep this in mind!

Concentration Inequalities

- The expression

$$\Pr[\, X - \mu \geq \mu \,]$$

  seems like a perfect case to try to use a concentration bound, like we did last Thursday.

- Knowing nothing about X other than the fact that it's nonnegative, we could start off by trying to use Markov's inequality:

$$\Pr[\, X \geq c\mathrm{E}[X] \,] \leq 1/c$$

- Using what we have:

$$\Pr[\, X - \mu \geq \mu \,] = \Pr[\, X \geq 2\mu \,] \leq ½.$$

- That's a pretty weak bound. What does that do to our analysis?

A Runtime Bound

- The expected cost of looking up x_q in a linear probing table is

$$O(1) \cdot \sum_{l=0}^{O(\log n)} 2^l \Pr[\text{the } 2^l \text{ block above } h(x_q) \text{ is bad}]$$

- Assuming 2-independent hashing, this is

$$\begin{aligned} & O(1) \cdot \sum_{l=0}^{O(\log n)} 2^l \Pr[\text{the } 2^l \text{ block above } h(x_q) \text{ is bad}] \\ \leq\ & O(1) \cdot \sum_{l=0}^{O(\log n)} 2^l \cdot \frac{1}{2} \\ =\ & O(1) \cdot \sum_{l=0}^{O(\log n)} 2^{l-1} \\ =\ & O(n) \end{aligned}$$

- This bound is not at all useful. We're going to need to do better than this!

Concentration Inequalities

- Our previous analysis of X used Markov's inequality, which makes no assumptions at all about the distribution of X. Let's see if we can use our knowledge of where X comes from to tighten the bound.
- Let X_i be an indicator variable that's 1 if x_i hashes into this block and 0 otherwise. Then we can write

$$X = \sum_{i=1}^{n} X_i.$$

- For notational simplicity $p_i = \mathrm{E}[X_i]$. We can evaluate this directly if we'd like, but it turns out that we won't actually be needing it.
- Notice that

$$\mu = E[X] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} p_i.$$

Chernoff Bounds

- Last time, we saw the ***Chernoff bound***, which says that if $X \sim \text{Binom}(n, p)$ and $p < 1/2$, then

$$\Pr[X > n/2] < e^{\frac{-n(1/2-p)^2}{2p}}$$

- We just saw that our variable X is the sum of a number of Bernoulli variables X_i, so it seems like we might be able to apply Chernoff bounds here.
- ***Problem:*** These X_i variables are ***not*** independent of one another! We're assuming that we have a k-independent hash function and are conditioning on knowing where x_q is, so while we can say that any $k - 1$ of these X_i's are independent, we can't say that any n of them are.
- Therefore, X is not binomially distributed, so we can't use a Chernoff bound.

Chebyshev's Inequality

- The last remaining bound that we used last time was ***Chebyshev's inequality***, which states that

$$\Pr [\ |X - \mu| \geq c\sigma\ ] \leq 1 / c^2,$$

  where σ^2 is the variance of X.

- If we can determine σ^2, then we can try using Chebyshev's inequality to bound the probability that X is a bad block.

The Variance

- For each X_i, let $\sigma_i^2 = \text{Var}[X_i]$. Then

$$\begin{aligned} \sigma^2 &= \text{Var}[X] \\ &= \text{Var}\left[\sum_{i=1}^{n} X_i\right] \end{aligned}$$

Assume, going forward, that the X_i's are pairwise independent.

We're already conditioning on knowing $h(x_q)$.

This means that we need our hash function to be at least **3-independent** from this point onward.

The Variance

- For each X_i, let $\sigma_i^2 = \text{Var}[X_i]$. Then

$$\begin{aligned}
\sigma^2 &= \text{Var}[X] \\
&= \text{Var}[\sum_{i=1}^{n} X_i] \\
&= \sum_{i=1}^{n} \text{Var}[X_i] \\
&= \sum_{i=1}^{n} \sigma_i^2 \\
&= \sum_{i=1}^{n} p_i(1-p_i) \\
&\leq \sum_{i=1}^{n} p_i \\
&= \mu
\end{aligned}$$

Using Chebyshev

- We want to know

$$\Pr[\, X - \mu \geq \mu \,].$$

- Using Chebyshev's inequality:

$$\begin{aligned} \Pr[X - \mu \geq \mu] &\leq \Pr[|X - \mu| \geq \mu] \\ &= \Pr[|X - \mu| \geq \sqrt{\mu}\sqrt{\mu}] \\ &\leq \Pr[|X - \mu| \geq \sqrt{\mu}\,\sigma] \\ &\leq \frac{1}{\mu} \end{aligned}$$

- So the probability that a block of length 2^l is bad is at most $\mu^{-1} = \mathbf{3 \cdot 2^{-l}}$.

A Better Bound

- The expected cost of looking up x_q in a linear probing table is

$$O(1) \cdot \sum_{l=0}^{O(\log n)} 2^l \Pr[\text{the } 2^l \text{ block above } h(x_q) \text{ is bad}]$$

- Assuming 3-independent hashing, this is

$$\begin{aligned} & O(1) \cdot \sum_{l=0}^{O(\log n)} 2^l \Pr[\text{the } 2^l \text{ block above } h(x_q) \text{ is bad}] \\ \leq\ & O(1) \cdot \sum_{l=0}^{O(\log n)} 2^l \cdot 3 \cdot 2^{-l} \\ =\ & O(1) \cdot \sum_{l=0}^{O(\log n)} 3 \\ =\ & O(\log n) \end{aligned}$$

- ***Theorem:*** This runtime bound is tight (there's an adversarial choice of a 3-independent hash function that degrades the runtime to this level.)

Why This Works

- ***Key idea:*** Increasing the degree of independence lets us control the variance of the distribution.
- With 2-independent hashing, we use one degree of independence to condition on knowing where some specific key lands. At that point, we only have one more degree of independence – not enough to control the variance!
- With 3-independent hashing, we use one degree of independence to condition on knowing where the key lands. We can then use the two remaining degrees of independence to control the variance and use Chebyshev's inequality.
- ***Small increases to the independence of a hash function can dramatically tighten concentration bounds.***

Question: If we increase the degree of independence further, can we constrain the spread of the elements in a way that improves our runtime?

(This is the theory version of “can we do better?”)

A Review: Deriving Chebyshev

- Let's take a minute to derive Chebyshev's inequality.
- Let X be a random variable with mean μ and variance σ^2. Then

$$\Pr[|X - \mu| \geq c\sigma] = \Pr[(X - \mu)^2 \geq c^2\sigma^2].$$

- Let $Y = (X - \mu)^2$. Notice that

$$\mathrm{E}[Y] = \mathrm{E}[(X - \mu)^2] = \sigma^2.$$

- So, via Markov's inequality, we have

$$\begin{aligned} \Pr[|X-\mu| \geq c\sigma] &= \Pr[(X-\mu)^2 \geq c^2\sigma^2] \\ &= \Pr[Y \geq c^2\sigma^2] \\ &= \Pr[Y \geq c^2\mathrm{E}[Y]] \\ &\leq \frac{1}{c^2}. \end{aligned}$$

Generalizing Chebyshev

- The variance of a random variable X with mean μ is defined as

$$\sigma^2 = E[(X - \mu)^2].$$

- We can generalize this to higher exponents.
- The ***kth central moment*** of a random variable X with mean μ, denoted μ_k, is defined as

$$\mu_k = E[(X - \mu)^k].$$

- Notice that

$$\mu_1 = E[(X - \mu)^1] = 0 \qquad \mu_2 = E[(X - \mu)^2] = \sigma^2$$

- Central moments give a way of measuring how much X is spread out from its mean. Higher central moments give progressively more weight to outliers.

The Fourth Moment Inequality

- The ***fourth moment inequality*** states that

$$\Pr[|X-\mu| \geq c\sqrt[4]{\mu_4}] \leq \frac{1}{c^4}$$

- It's a generalization of Chebyshev's inequality, and the proof is similar.
- Let X be a random variable with mean μ and fourth moment μ_4. Then

$$\Pr[|X-\mu| \geq c\sqrt[4]{\mu_4}] = \Pr[(X-\mu)^4 \geq c^4\mu_4]$$

- Let $Y = (X - \mu)^4$. Notice that

$$E[Y] = E[(X - \mu)^4] = \mu_4.$$

- So, via Markov's inequality, we have

$$\begin{aligned}\Pr[|X-\mu| \geq c\sqrt[4]{\mu_4}] &= \Pr[(X-\mu)^4 \geq c^4\mu_4] \\ &= \Pr[Y \geq c^4\mu_4] \\ &= \Pr[Y \geq c^4 E[Y]] \\ &\leq \frac{1}{c^4}.\end{aligned}$$

Good question to ponder:
why doesn't this work for the third central moment?

Updating our Analysis

- For linear probing, we're ultimately interested in bounding

$$\Pr[\, X - \mu \geq \mu \,]$$

  in the case where X represents the number of elements hitting a particular block.
- Using 2-independent hashing, the best bound we could use was Markov's inequality, which gave an extremely weak bound.
- Using 3-independent hashing, we could use Chebyshev's inequality, which gave us a bound of μ^{-1}.
- ***Question:*** If we use stronger hash functions, can we tighten this bound using the fourth moment inequality?

The Fourth Moment

- Let $\mu_4 = E[(X - \mu)^4]$. We want to get a nice bound on μ_4.

$$
\begin{aligned}
E[(X-\mu)^4] &= E\left[\left(\sum_{i=1}^{n} X_i - \sum_{i=1}^{n} p_i\right)^4\right] \\
&= E\left[\left(\sum_{i=1}^{n} (X_i - p_i)\right)^4\right] \\
&= E\left[\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n} (X_i-p_i)(X_j-p_j)(X_k-p_k)(X_l-p_l)\right] \\
&= \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n} E[(X_i-p_i)(X_j-p_j)(X_k-p_k)(X_l-p_l)]
\end{aligned}
$$

- So now we “just” need to simplify this expression.

Increasing our Independence

$$\mathrm{E}[(X-\mu)^4] = \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n}\mathrm{E}[(X_i-p_i)(X_j-p_j)(X_k-p_k)(X_l-p_l)]$$

- ***Recall:*** If our hash function is k-independent, then we've already used one degree of independence conditioning on knowing where $h(x_q)$ is. That leaves us with k-1 degrees of independence.
- Let's suppose we're using a **5-independent** hash function, meaning that any four hash values are independent of one another.
- This allows us to make some amount of progress in simplifying this expression.

Exploring this Summation

- The terms of this summation might sometimes range over the same variables at the same time:

$$\mathrm{E}[(X-\mu)^4] = \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n}\mathrm{E}[(X_i-p_i)(X_j-p_j)(X_k-p_k)(X_l-p_l)]$$

- ***Claim:*** Any term in the summation where $i \neq j$, $i \neq k$, and $i \neq l$ is 0.

- ***Proof:*** Suppose that X_i is a different random variable from the others. Then

$$\begin{aligned} & \mathrm{E}[(X_i-p_i)(X_j-p_j)(X_k-p_k)(X_l-p_l)] \\ = \; & \mathrm{E}[X_i-p_i]\mathrm{E}[(X_j-p_j)(X_k-p_k)(X_l-p_l)] \\ = \; & 0\cdot\mathrm{E}[(X_j-p_j)(X_k-p_k)(X_l-p_l)] \\ = \; & 0 \end{aligned}$$

Exploring this Summation

- The terms of this summation might sometimes range over the same variables at the same time:

$$\mathrm{E}[(X-\mu)^4] = \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n}\mathrm{E}[(X_i-p_i)(X_j-p_j)(X_k-p_k)(X_l-p_l)]$$

- ***Claim:*** The above summation reduces only to the case where $i=j=k=l$ and the case where there are exactly two distinct random variables in the product.
- ***Proof:*** If a variable appears exactly once, it doesn't contribute to the total. If a variable appears three times, some other variable contributes exactly once.

$$\mathrm{E}[(X-\mu)^4] = \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n}\mathrm{E}[(X_i-p_i)(X_j-p_j)(X_k-p_k)(X_l-p_l)]$$

$$= \sum_{i=1}^{n}\mathrm{E}[(X_i-p_i)^4] + \binom{4}{2}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathrm{E}[(X_i-p_i)^2(X_j-p_j)^2]$$

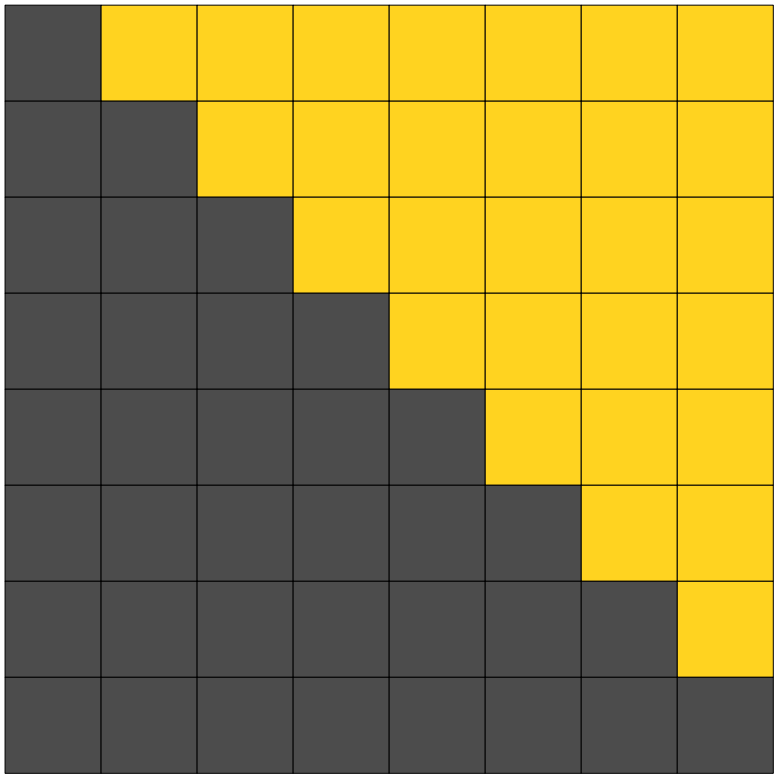
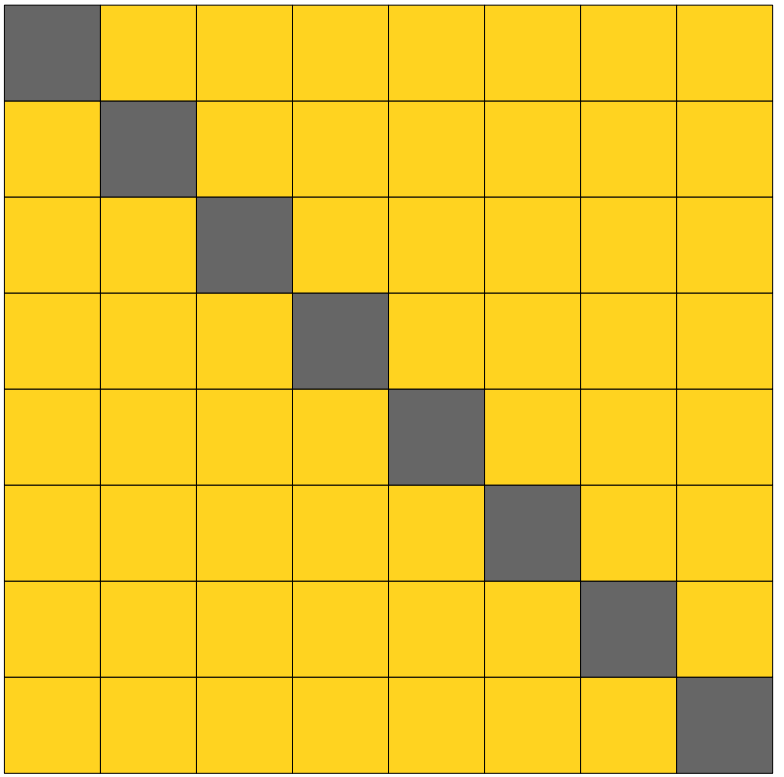
This term represents all the possible ways we could pick something four times.

$$\mathrm{E}[(X-\mu)^4] = \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n}\mathrm{E}[(X_i-p_i)(X_j-p_j)(X_k-p_k)(X_l-p_l)]$$

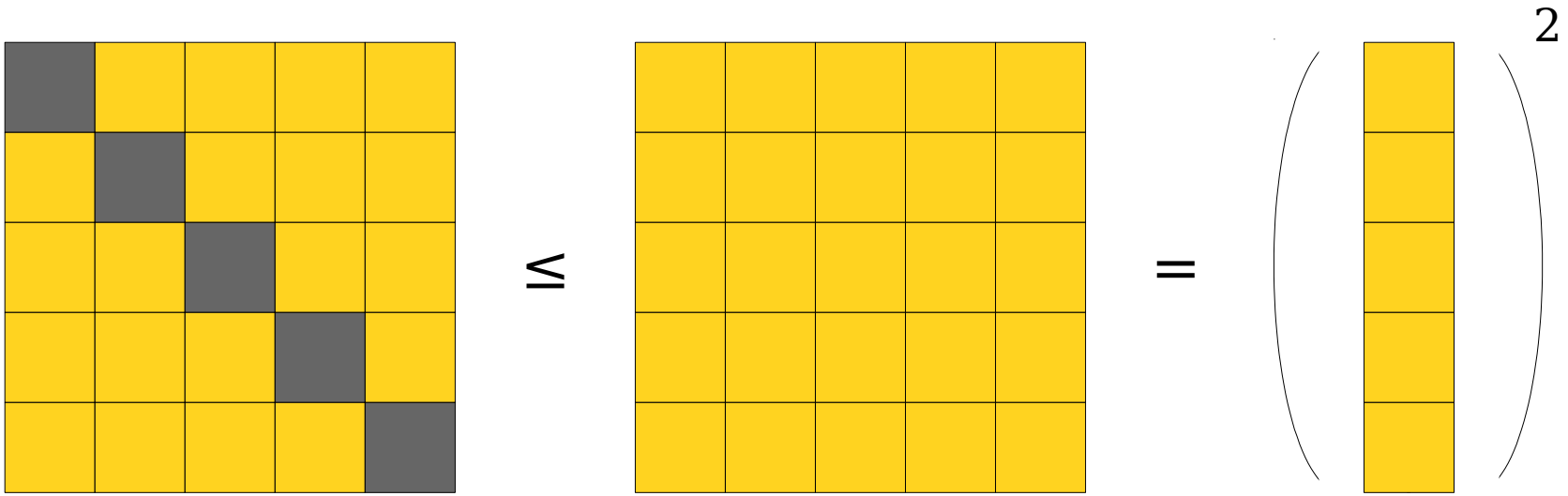
$$= \sum_{i=1}^{n}\mathrm{E}[(X_i-p_i)^4] + \binom{4}{2}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathrm{E}[(X_i-p_i)^2(X_j-p_j)^2]$$

This accounts for the two-and-two case. We choose two of the four indices to serve as *i* and the other two to serve as *j*, hence the coefficient.

$$
\begin{aligned}
\mathrm{E}[(X-\mu)^4] &= \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n}\mathrm{E}[(X_i-p_i)(X_j-p_j)(X_k-p_k)(X_l-p_l)] \\
&= \sum_{i=1}^{n}\mathrm{E}[(X_i-p_i)^4] + \binom{4}{2}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathrm{E}[(X_i-p_i)^2(X_j-p_j)^2] \\
&= \sum_{i=1}^{n}\mathrm{E}[(X_i-p_i)^4] + 6\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathrm{E}[(X_i-p_i)^2]\mathrm{E}[(X_j-p_j)^2] \\
&= \sum_{i=1}^{n}\mathrm{E}[(X_i-p_i)^4] + 6\sum_{i=1}^{n}\sum_{j=i+1}^{n}\sigma_i^2\sigma_j^2
\end{aligned}
$$

$6 \times$  $= 3 \times$ 

$$
\begin{aligned}
\mathrm{E}[(X-\mu)^4] &= \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n}\mathrm{E}[(X_i-p_i)(X_j-p_j)(X_k-p_k)(X_l-p_l)] \\
&= \sum_{i=1}^{n}\mathrm{E}[(X_i-p_i)^4] + \binom{4}{2}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathrm{E}[(X_i-p_i)^2(X_j-p_j)^2] \\
&= \sum_{i=1}^{n}\mathrm{E}[(X_i-p_i)^4] + 6\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathrm{E}[(X_i-p_i)^2]\mathrm{E}[(X_j-p_j)^2] \\
&= \sum_{i=1}^{n}\mathrm{E}[(X_i-p_i)^4] + 6\sum_{i=1}^{n}\sum_{j=i+1}^{n}\sigma_i^2\sigma_j^2 \\
&= \sum_{i=1}^{n}\mathrm{E}[(X_i-p_i)^4] + 3\sum_{i\neq j}\sigma_i^2\sigma_j^2
\end{aligned}
$$



$$
\begin{aligned}
\mathrm{E}[(X-\mu)^4] &= \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n}\mathrm{E}[(X_i-p_i)(X_j-p_j)(X_k-p_k)(X_l-p_l)] \\
&= \sum_{i=1}^{n}\mathrm{E}[(X_i-p_i)^4] + \binom{4}{2}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathrm{E}[(X_i-p_i)^2(X_j-p_j)^2] \\
&= \sum_{i=1}^{n}\mathrm{E}[(X_i-p_i)^4] + 6\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathrm{E}[(X_i-p_i)^2]\mathrm{E}[(X_j-p_j)^2] \\
&= \sum_{i=1}^{n}\mathrm{E}[(X_i-p_i)^4] + 6\sum_{i=1}^{n}\sum_{j=i+1}^{n}\sigma_i^2\sigma_j^2 \\
&= \sum_{i=1}^{n}\mathrm{E}[(X_i-p_i)^4] + 3\sum_{i\neq j}\sigma_i^2\sigma_j^2 \\
&\leq \sum_{i=1}^{n}\mathrm{E}[(X_i-p_i)^4] + 3\left(\sum_{i=1}^{n}\sigma_i^2\right)^2 \\
&= \sum_{i=1}^{n}\mathrm{E}[(X_i-p_i)^4] + 3(\sigma^2)^2 \\
&= \sum_{i=1}^{n}\mathrm{E}[(X_i-p_i)^4] + 3\sigma^4 \\
&\leq \sigma^2 + 3\sigma^4
\end{aligned}
$$

Fact: The fourth central moment of a Bernoulli random variable with probability p_i is at most its variance.
(Prove this!)

The Net Result

- We've just shown that

$$\mu_4 \leq \sigma^2 + 3\sigma^4.$$

- Since $\sigma^2 \leq \mu$, we can upper-bound this at

$$\mu_4 \leq \mu + 3\mu^2.$$

- For a sufficiently large block size, we know that $\mu \geq 1$. Under that assumption, we get that

$$\mu_4 \leq \mu^2 + 3\mu^2 = 4\mu^2.$$

- Phew! That was crazy. But at least we now have a bound on the fourth moment, which lets us use the fourth moment inequality!

Fourth Moments for Victory

- Using the fourth moment inequality:

$$\begin{aligned}
\Pr[X - \mu \geq \mu] &\leq \Pr[|X - \mu| \geq \mu] \\
&= \Pr\left[|X - \mu| \geq \frac{\sqrt{\mu}}{\sqrt{2}}\sqrt{2\mu}\right] \\
&\leq \Pr\left[|X - \mu| \geq \frac{\sqrt{\mu}}{\sqrt{2}}\sqrt[4]{\mu_4}\right] \\
&\leq \left(\frac{\sqrt{\mu}}{\sqrt{2}}\right)^{-4} \\
&= \frac{4}{\mu^2}
\end{aligned}$$

- So the probability that a block of length 2^l is bad is at most $4\mu^{-2} = \mathbf{36 \cdot 2^{-2l}}$.
- Notice that this is exponentially better than our previous bound!

A Strong Runtime Bound

- The expected cost of looking up x_q in a linear probing table is

$$O(1) \cdot \sum_{l=0}^{O(\log n)} 2^l \Pr[\text{the } 2^l \text{ block above } h(x_q) \text{ is bad}]$$

- Assuming 5-independent hashing, this is

$$\begin{aligned} & O(1) \cdot \sum_{l=0}^{O(\log n)} 2^l \Pr[\text{the } 2^l \text{ block above } h(x_q) \text{ is bad}] \\ = \; & O(1) \cdot \sum_{l=0}^{O(\log n)} 2^l \cdot 36 \cdot 2^{-2l} \\ = \; & O(1) \cdot \sum_{l=0}^{O(\log n)} 2^{-l} \\ = \; & O(1) \end{aligned}$$

- We've finally obtained an O(1) bound on the cost of operations in a chained hash table – provided that we use 5-independent hashing!

What Just Happened?

- Ultimately, we wanted to bound the probability that a block was twice as full as its expectation.
- With one degree of independence, we could obtain the *expected value* and use that to bound the probability with Markov's inequality.
- Using two degrees of independence, we could obtain the *variance* and use that to bound the probability with *Chebyshev's inequality*.
- Using four degrees of independence, we could obtain the *fourth central moment* and use that to bound the probability with the *fourth moment bound*.
- Increasing the strength of a hash function allows us to obtain more central moments and, therefore, to tighten our bound more than might initially be suspected.
- This technique shows up a *lot* in randomized data structures. You'll see it appear in the tug-of-war sketch on Problem Set Five.

Some Concluding Notes

Harnessing Entropy

- In our analysis, we made no assumptions about what specific keys we were placing in our hash table.
- ***Theorem (Mitzenmacher and Vadhan):*** Using 2-independent hash functions, if there is a reasonable amount of entropy in the distribution of the keys, linear probing takes time O(1).
- The proof essentially shows that 2-universal hash functions applied to data with sufficient entropy very closely approximate truly random functions.
- Intuitively, they get the “missing” randomness from the distribution rather than from the hash function itself.
- See “Why Simple Hash Functions Work” for details – it's a remarkable argument!

Next Time

- ***Cuckoo Hashing***
  - Hashing with worst-case O(1) lookups!
- ***The Cuckoo Graph***
  - Random graphs for Fun and Profit.