

CS848 Fall 2026: Indexing for Modern Data Processing

Hashing

Xiao Hu

September 17, 2026

Announcement

- Please sign up course presentation by this week!
 - Course schedule will be finalized next week
 - Extra presentation slots: If you would like to present a second topic, please feel free to sign up and gain bonus. Students auditing the class are also welcome to contribute and present!
- Next lecture (Sep 22) is online
 - We won't meet in DC 2568
 - Zoom meeting link is given on LEARN
- Next Next lecture (Sep 24) is back to DC2568
 - Great thanks to Tom Iagovet for a 20min presentation on concurrent B+ tree

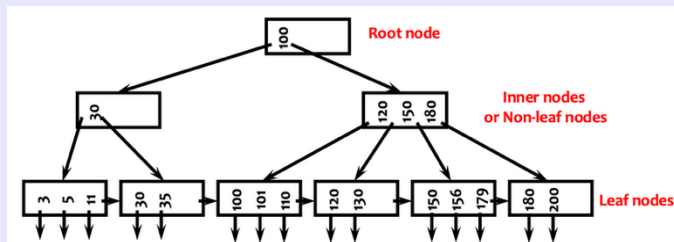
Roadmap

- Hashing
- Dynamic Hashing: Extendible hashing and Linear hashing
- Good Hashing Functions
 - Universal hashing
 - k-wise independent hashing
 - Cuckoo hashing
- Where hashing appears in modern systems

Recap from last lecture

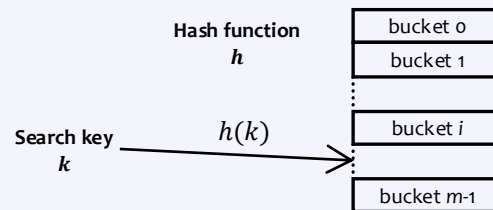
Maintain a set of keys (from domain U) under insert, delete, and lookup.

B+ tree index



Keys stay ordered in the structure.
Lookup costs $O(\log_f N)$ node reads.
Supports range scans and ORDER BY.

Hash index



A hash function computes the bucket address directly.
No order, so no range support.

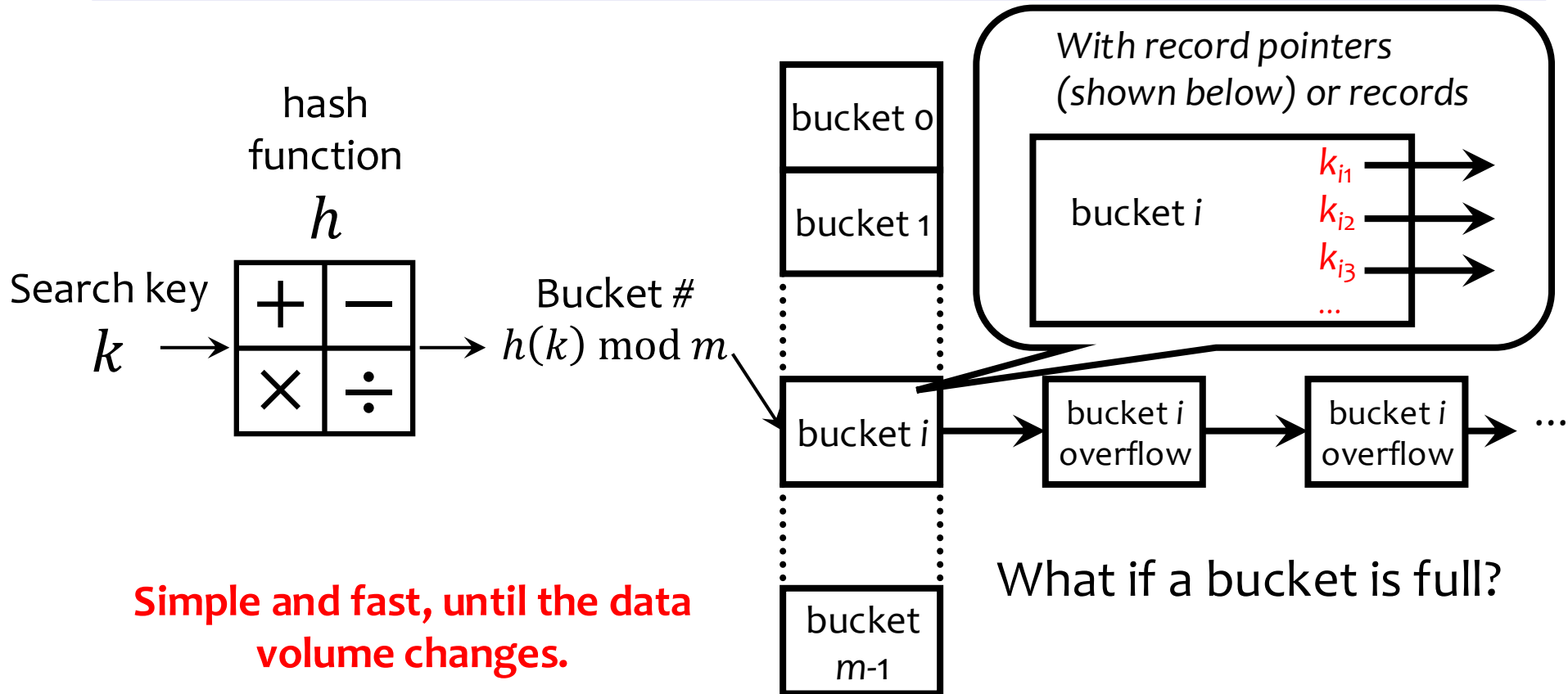
Hashing trades order for direct addressing.

Terminology of a Hash Index

Hash function h	Maps a key to a bucket number in $\{0, \dots, m-1\}$.
Bucket	One disk page holding entries (and additional pointers to overflow pages)
Load factor	N/m , the average number of keys per bucket.
Collision	Two distinct keys landing in the same bucket.

Design questions: which h , and what to do on collision.

Static Hashing



Simple and fast, until the data volume changes.

What if a bucket is full?

Performance of Static Hashing

- Best case is one I/O per equality lookup. That is the reason to use a hash index.
- Average case depends on the quality of h and on the load factor.
- Worst case sends all keys to one bucket, so a lookup scans all data
- How to deal with the growth problem?
 - Rehash the whole table — using a new hash function, or at least changing m to the new number of buckets
 - Entries in an old bucket may show up across many different new buckets, causing lots of I/Os
 - Cost of (re)building a giant hash table on disk $\approx$ sorting
- Is it possible to reduce data movement?

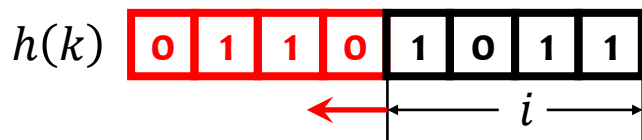
Target of Dynamic Hashing

- 1 Equality lookup should stay at one or two block reads as the file grows.
- 2 Growth should touch a bounded number of blocks, not the whole index.
- 3 Space overhead should stay proportional to the data size.

Extendible Hashing and Linear Hashing are two different methods!

Extendible Hashing

- **Idea 1:** have hash function ***h*** output a large number of bits, but only use the lowest i bits, and dynamically increase i as needed

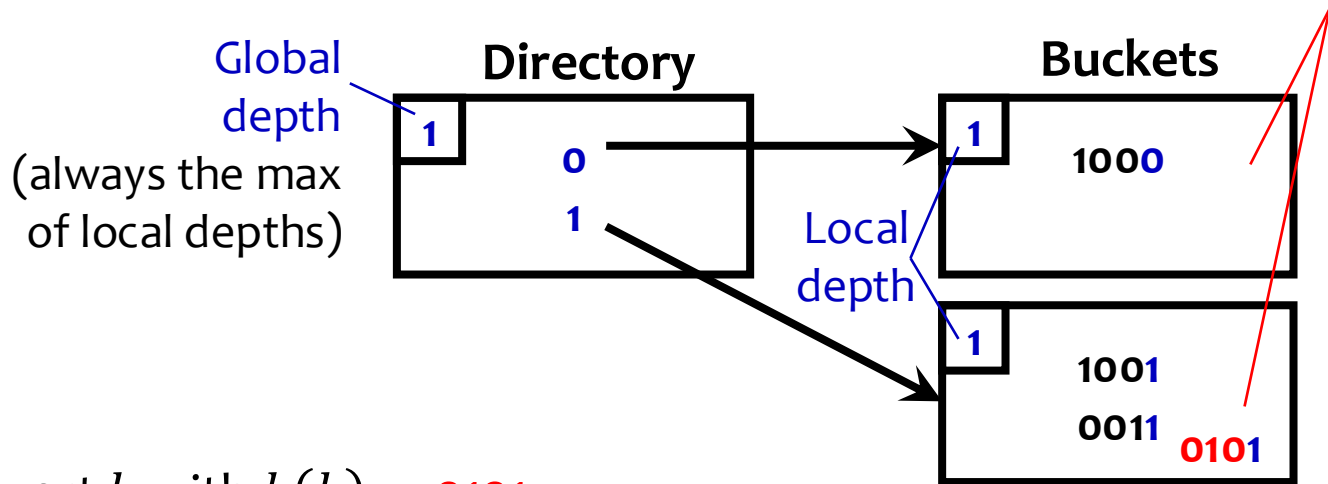


- E.g., $h_4(k) = 1011$, then $h_5(k)$ is either **0**1011 or **1**1011 $\Rightarrow$ contents in one old bucket can only go to two new buckets!
- Problem: $++i$ doubles the number of buckets!
- **Idea 2:** use a directory
 - Only split overflowing buckets
 - But double the directory size
 - Many directory entries can point to the same bucket

Extendible Hashing Insertions

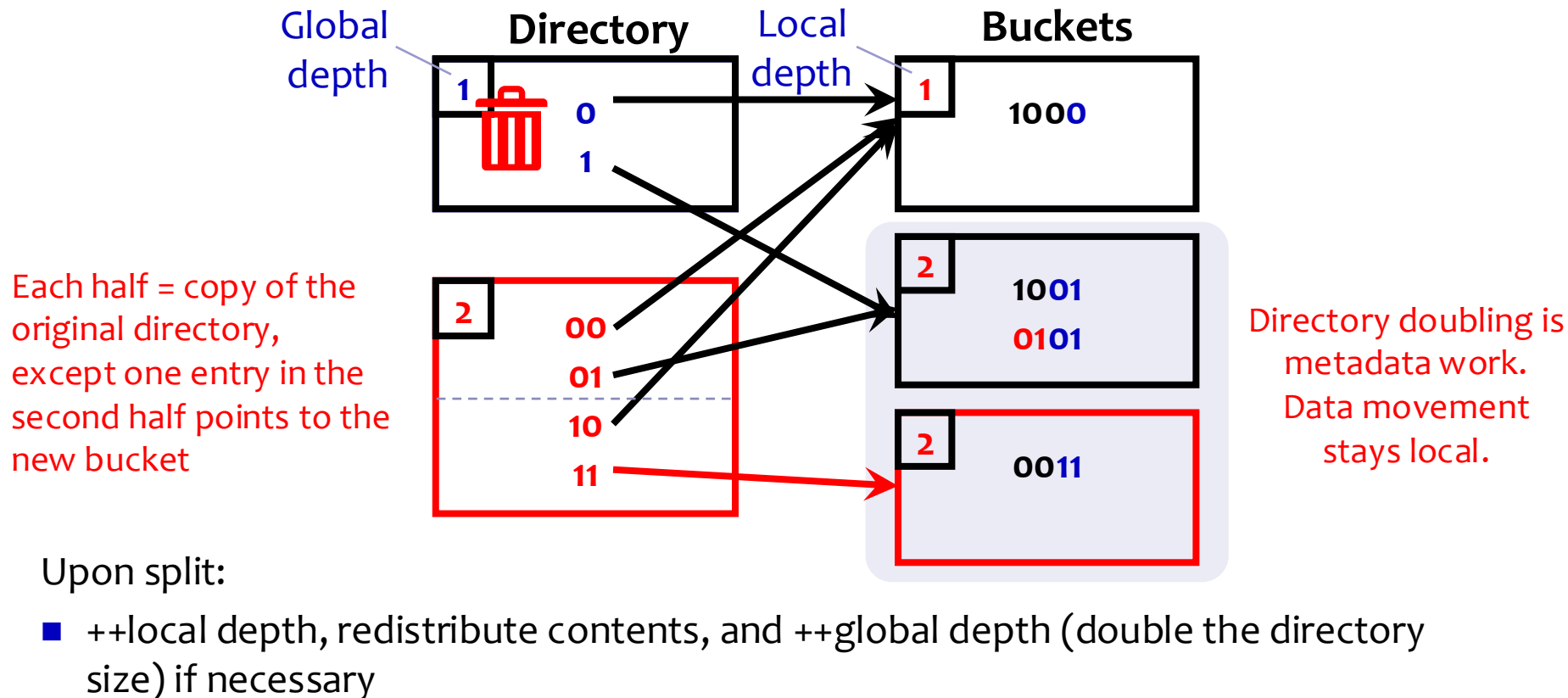
Assume each bucket can store at most 2 keys

Showing hashed values here for ease of understanding, but in reality, we store original key values



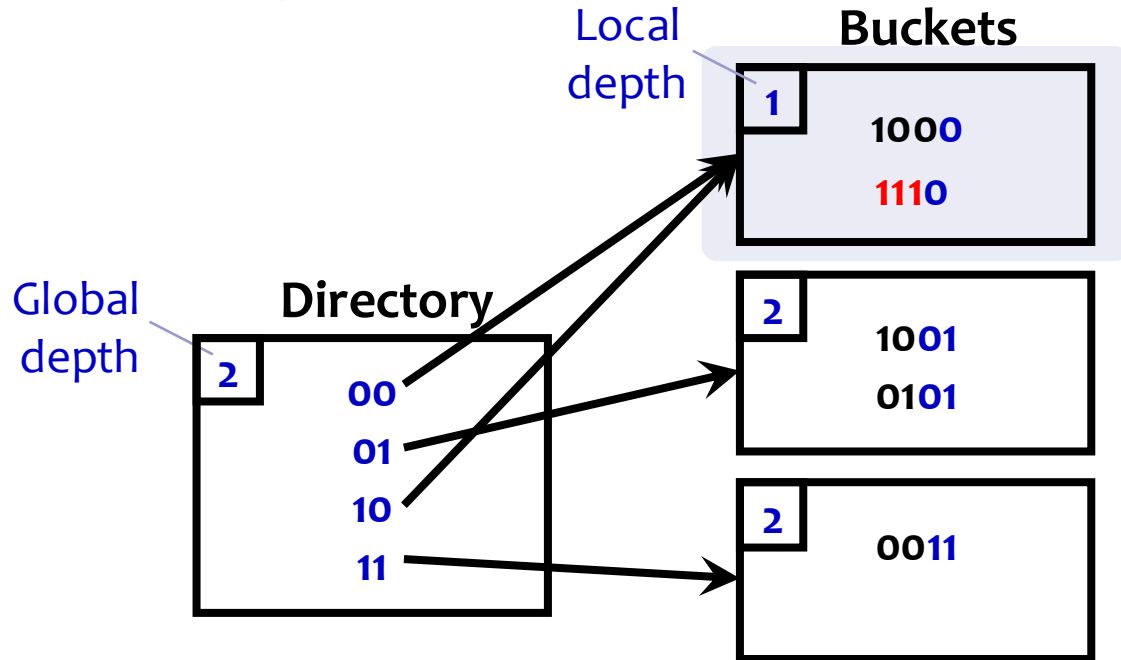
- Insert k with $h(k) = \mathbf{0101}$
- Bucket too full? **Split**
 - Allowing some overflow is also fine (and sometimes necessary)

Extendible Hashing Insertions



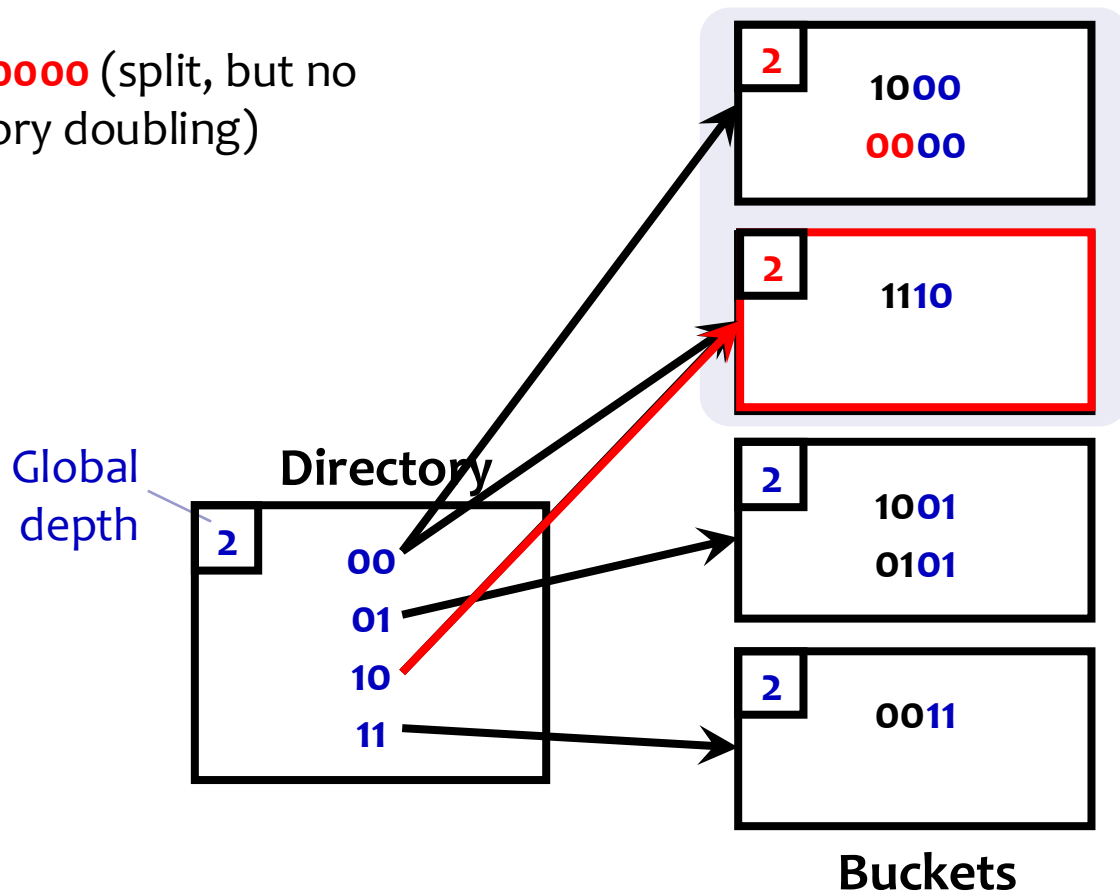
Extendible Hashing Insertions

- Insert **1110**
(no split necessary)



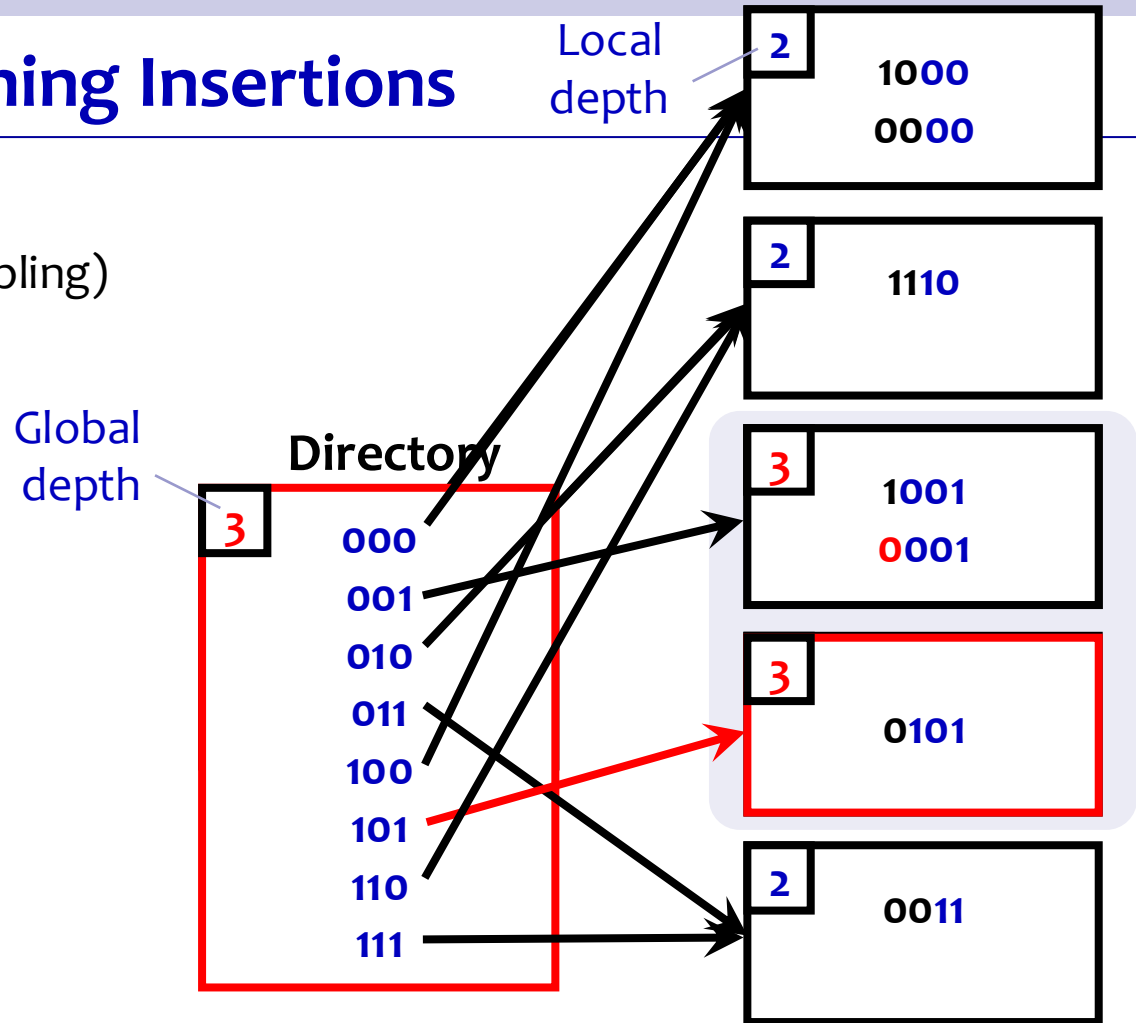
Extendible Hashing Insertions

- Insert **0000** (split, but no directory doubling)



Extendible Hashing Insertions

- Insert **0001**
(split + directory doubling)



Extendible Hashing Deletions

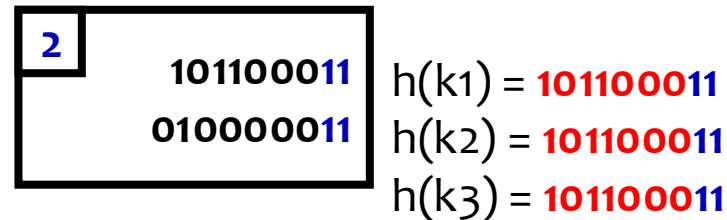
Just the reverse of insertions

- If the bucket becomes too empty:
 - Merge with “sibling” (differing only on the leading bit)
 - Adjust any pointer from the directory as needed
 - – – local depth
 - If possible, – – global depth and half the directory
 - Invariant: global depth = max of all local depths

Summary of Extensible Hashing

Pros:

- Handles growing/shrinking indexes
- No full reorganization
- One or two page reads per lookup (assuming the directory is cached in main memory; no overflow bucket)



Cons:

- One more level of indirection through the directory
- Directory size still doubles/halves
- There are cases when doubling may not be enough (see above)
- The directory can grow large under skewed hash distribution

Can you come up with a worst-case sequence of insertions?

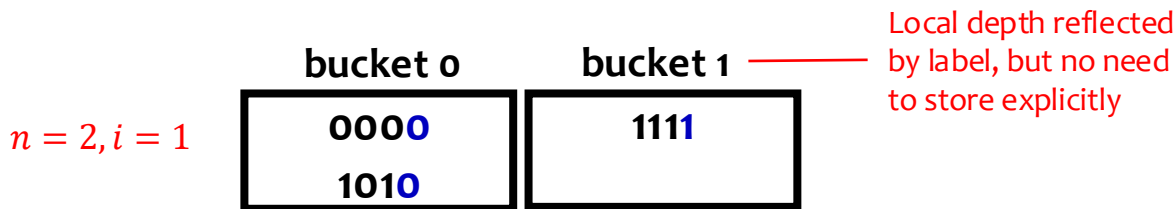
Linear hashing

- No extra indirection through a directory
 - Fix the splitting/growth order
 - Use some extra math to figure out the right bucket
- Grow only when **utilization** (avg. # entries per bucket / max # entries per block) exceeds a given threshold

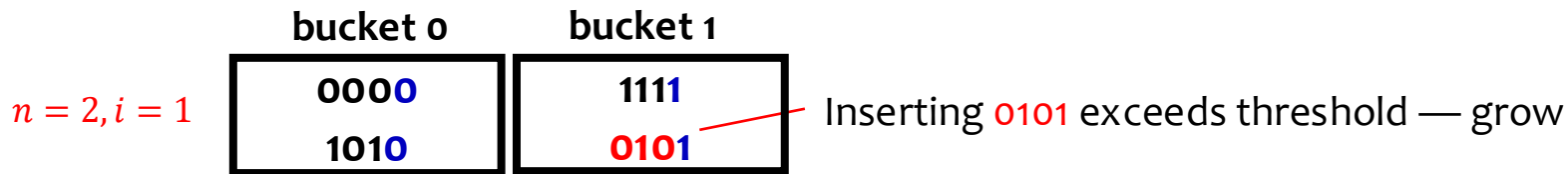
n : # of primary buckets (not counting overflow blocks)

$i = \lceil \log_2 n \rceil$: # of hash bits in use (global depth)

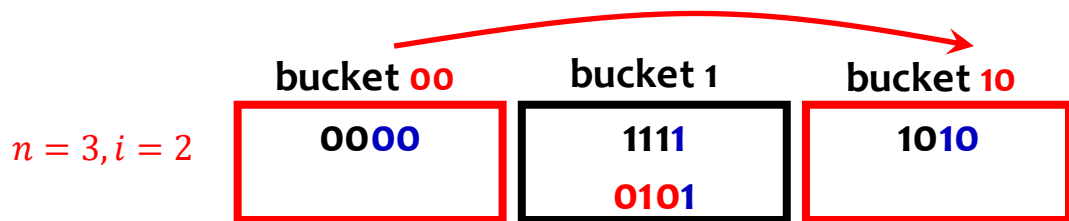
threshold = 85% (a range of the buckets may use $i - 1$ bits)



Linear hashing Insertion

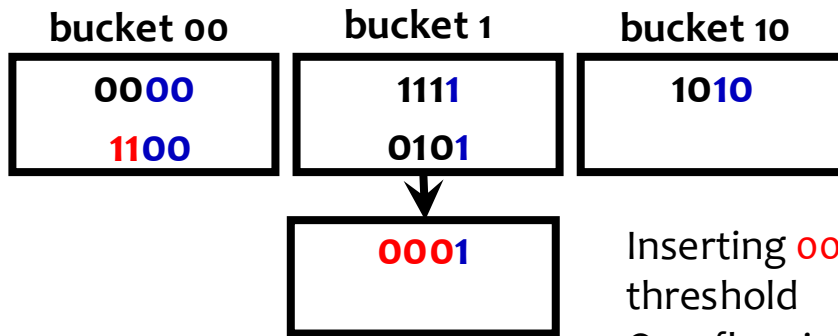


- Split the first bucket with the lowest depth — it's always the bucket $n - 2^{\lfloor \log_2 n \rfloor}$ (0-based index)
 - Often not the bucket you are inserting into!
- File grows *linearly* at the end (hence the name)



Linear hashing Insertion

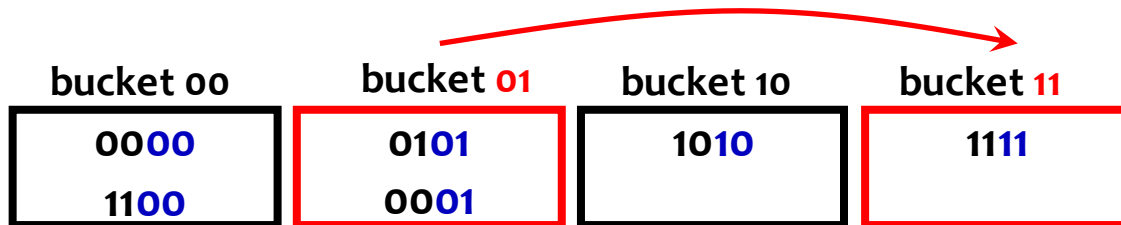
$n = 3, i = 2$



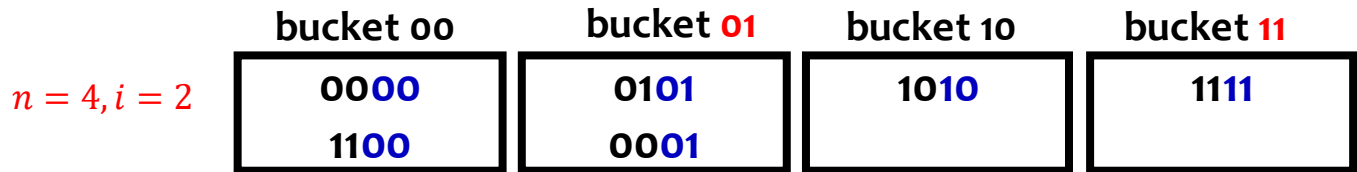
Inserting 0001 doesn't exceed threshold
Overflow is needed

Inserting 1100 exceeds threshold — grow

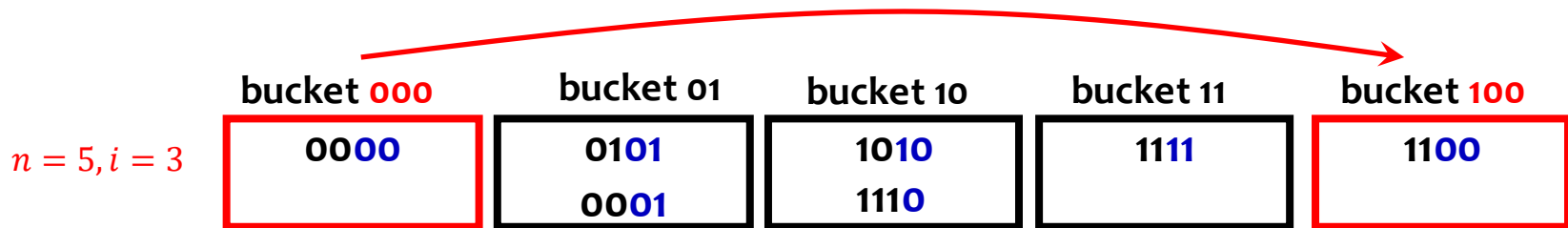
$n = 4, i = 2$



Linear hashing Insertion and Lookup



Inserting 1110 exceeds threshold — grow



Look up 1110: Bucket 110 doesn't exist, so go to bucket 10

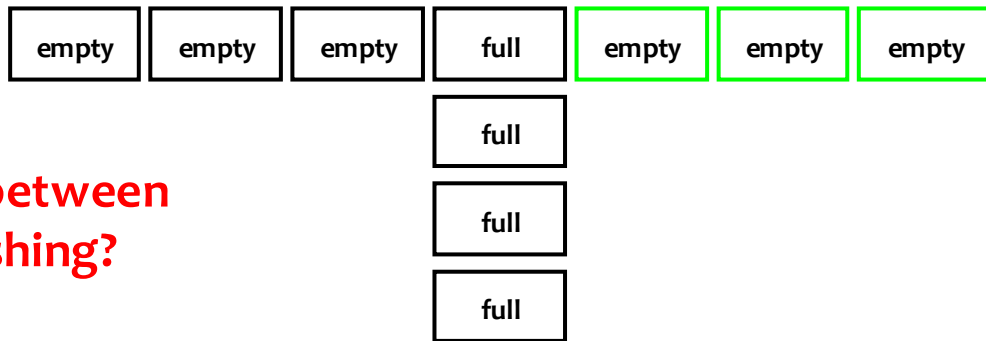
Summary of linear hashing

Pros:

- Handles growing/shrinking indexes
- No full reorganization
- No extra level of indirection (beats extensible hashing)

Cons:

- Still has overflow chains, and may not split them right away because buckets must be split in sequence



Can you summarize the difference between extensible hashing and linear hashing?

What Makes a Hash Function Good?

No Fixed Hash Function Is Safe

- **Claim:** for any fixed h from U to $\{0, \dots, m-1\}$ with $|U| \geq mN$, some set of N keys hashes to one slot.
 - If every slot received at most $N-1$ keys of U , then $|U| \leq m(N-1) < mN$.
 - An adversary that knows h can always build a worst-case workload.
- How to define a family H of hash functions, then sample h from H at random such that guarantees hold in expectation over the choice of h , for every input data?
- Unfortunately, truly random hashing family doesn't exist. (More precisely, such a function requires $O(|U| \log m)$ bits to represent.)

Can you see why?

Universal Hashing

- H is universal if for all distinct x, y in U , $\Pr[h(x) = h(y)] \leq 1/m$ when h is drawn at random from H
 - This is exactly the collision rate of a truly random function
 - It is far weaker than full independence, so cheap to build and store.

$U = \{a, b, c, d, e, f\}$
and $m = 2$.

	a	b	c	d	e	f
$h_1(x)$	0	1	0	1	0	1
$h_2(x)$	0	0	0	1	1	1

Is $H = \{h_1, h_2\}$ universal?

	a	b	c	d	e	f
$h_1(x)$	0	1	0	1	0	1
$h_2(x)$	0	0	0	1	1	1
$h_3(x)$	0	0	1	0	1	1
$h_4(x)$	1	0	0	1	1	0

Is $H = \{h_1, h_2, h_3, h_4\}$ universal?

Why Universal Hashing is good?

- **Theorem:** Let H be universal hashing family and $|S| = N$. For any key x and random h in H , the expected number of keys of S colliding with x is less than N/m .
 - Hence, if $N \leq m$, then the expected number of collisions with x is $O(1)$.
- The following H is universal
 - Choose a prime number p between $|U|$ and $2|U|$ (**Bertrand's Conjecture**)
 - $h_{a,b}(x) = ((ax + b) \bmod p) \bmod m$
 - $H = \{h_{a,b} : a, b \in \{0, 1, \dots, p-1\}, a \neq 0\}$

Can you prove this by analyzing #collisions?

Pairwise-Independent Hashing

- H is pairwise-independent if for all distinct x, y in U , the pair $(h(x), h(y))$ is uniform over all m^2 outcomes when h is drawn at random from H .
 - Pairwise-independence implies universality, but not the other way around.
- The following H is pairwise-independent
 - $h_{a,b}(x) = (ax + b) \bmod p$
 - $H = \{h_{a,b} : a, b \in \{0, 1, \dots, p-1\}\}$

Can you prove this by analyzing the probability distribution?

K-wise Independent Hashing

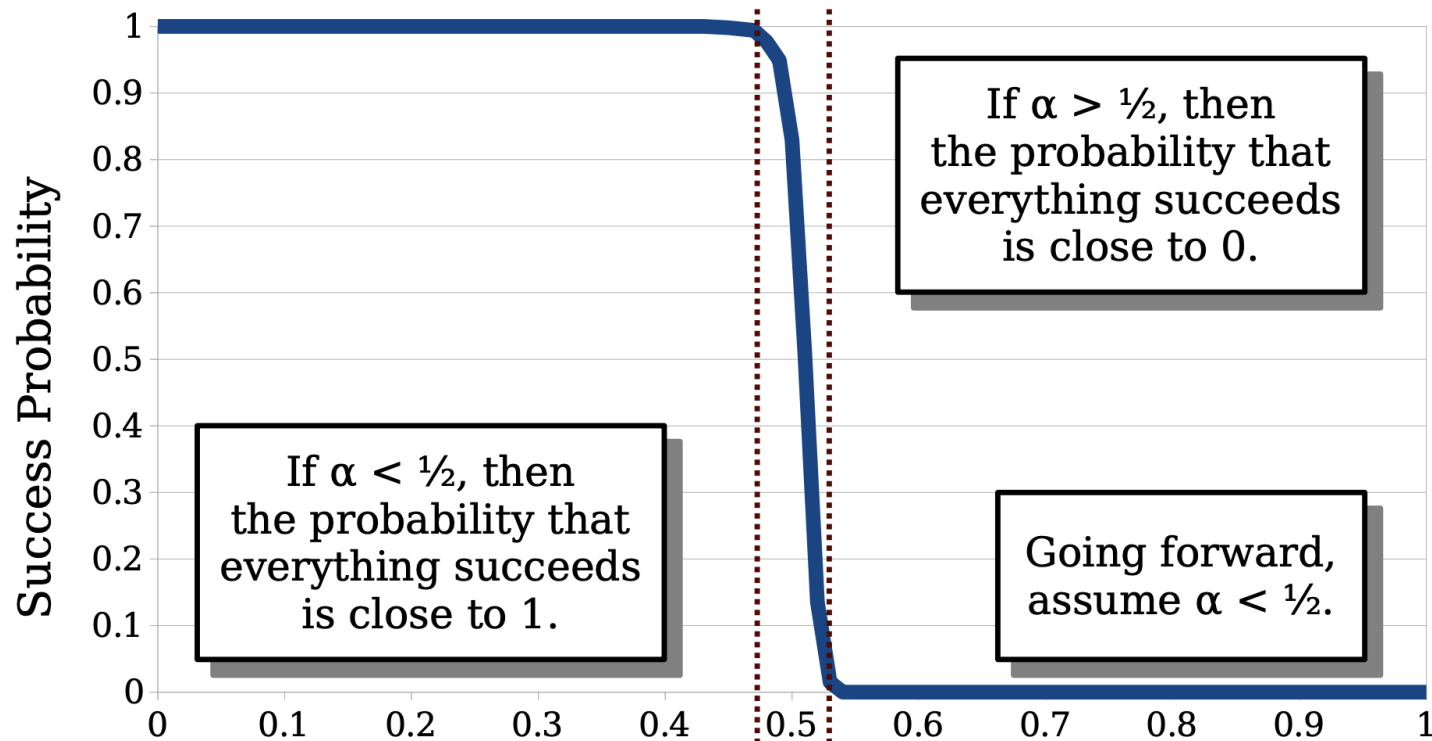
- H is k -wise independent if for k arbitrary distinct keys $x_1, x_2, \dots, x_k$ in U , the pair $(h(x_1), h(x_2), \dots, h(x_k))$ is uniform over all m^k outcomes when h is drawn at random from H .
- The following H is k -wise independent.
 - $h_{a_1, a_2, \dots, a_k}(x) = (a_1 x^{k-1} + a_2 x^{k-2} + \dots + a_{k-1} x + a_k) \bmod p$
 - $H = \{h_{a_1, a_2, \dots, a_k} : a_1, a_2, \dots, a_k \in \{0, 1, \dots, p-1\}\}$
- Higher k buys concentration bounds, not just expectations. (we will dive into this in later lectures)

Cuckoo Hashing: Worst-case Lookup

- **Idea:** Keep two tables and two hash functions h_1 and h_2
- **Invariant:** key x sits at slot $h_1(x)$ or at slot $h_2(x)$, nowhere else.
- Lookup and deletion check two slots, so both are worst-case $O(1)$.
- **Insertion:** $O(1)$ in expectation or amortized
 - To insert x into the table, first try placing it at slot $h_1(x)$.
 - If that slot is full, kick out the element y that used to be in that slot and try placing it the other slot it can belong to (either $h_1(y)$ or $h_2(y)$).
 - Repeat this process until all elements stabilize.
- **Failure:** If the chain of displacement goes to an infinite loop

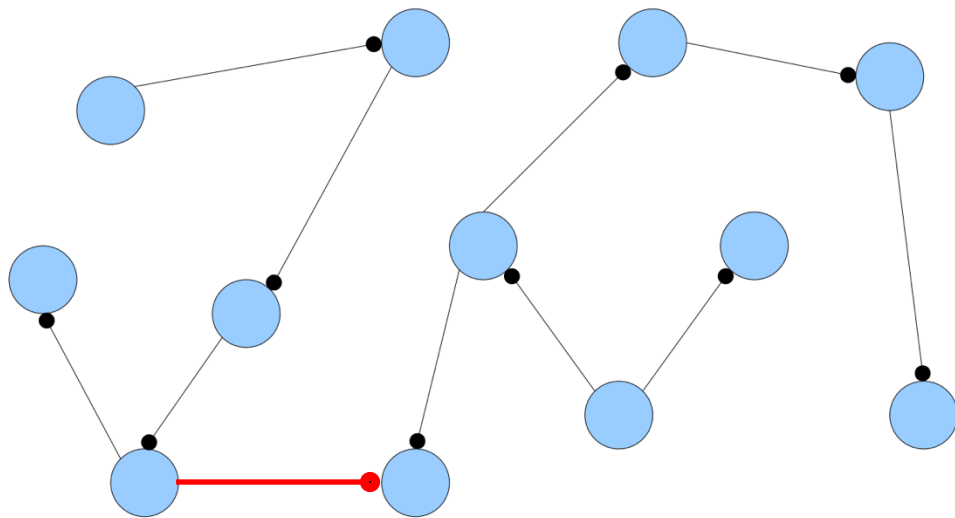
How Insertion Behaves in Practice

- Probability that all insertions succeed, as a function of the load factor $\alpha = \frac{N}{m}$?



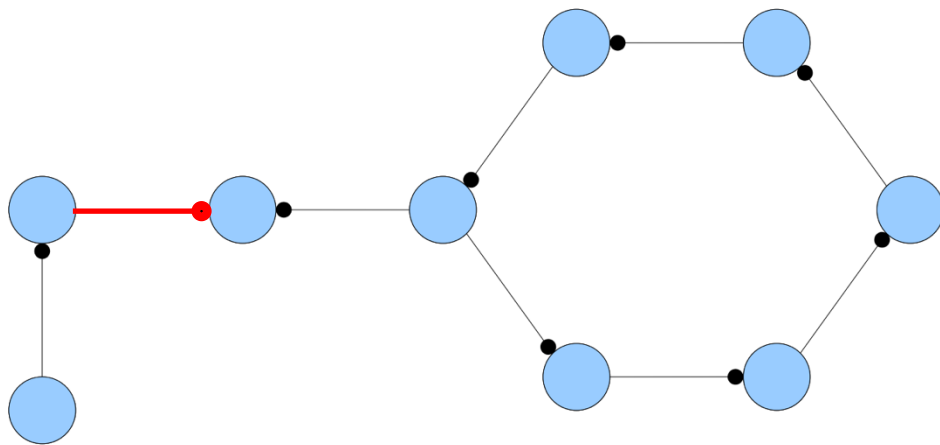
The Cuckoo Graph

- One node per slot, one edge per key joining its two candidate slots.
- Mark the endpoint where the key sits. Each node carries at most one mark.
- An insertion adds an edge; a displacement chain flips marks along a path.



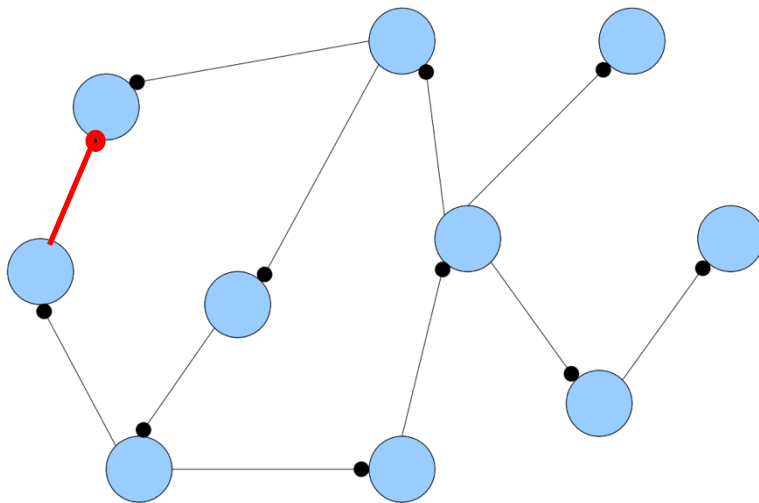
The Cuckoo Graph

- One node per slot, one edge per key joining its two candidate slots.
- Mark the endpoint where the key sits. Each node carries at most one mark.
- An insertion adds an edge; a displacement chain flips marks along a path.



The Cuckoo Graph

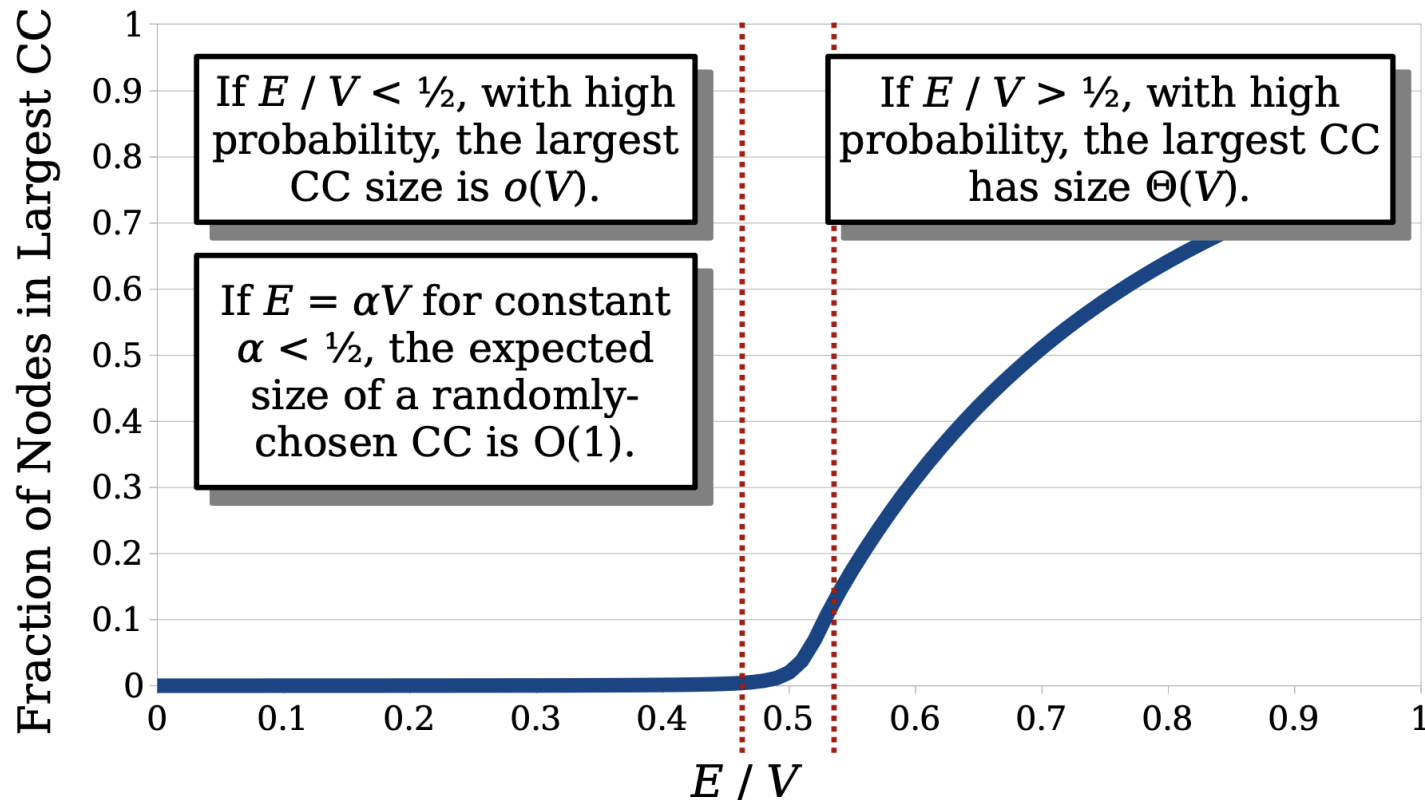
- An insertion of x fails if the connected component containing x contains more than one cycle.



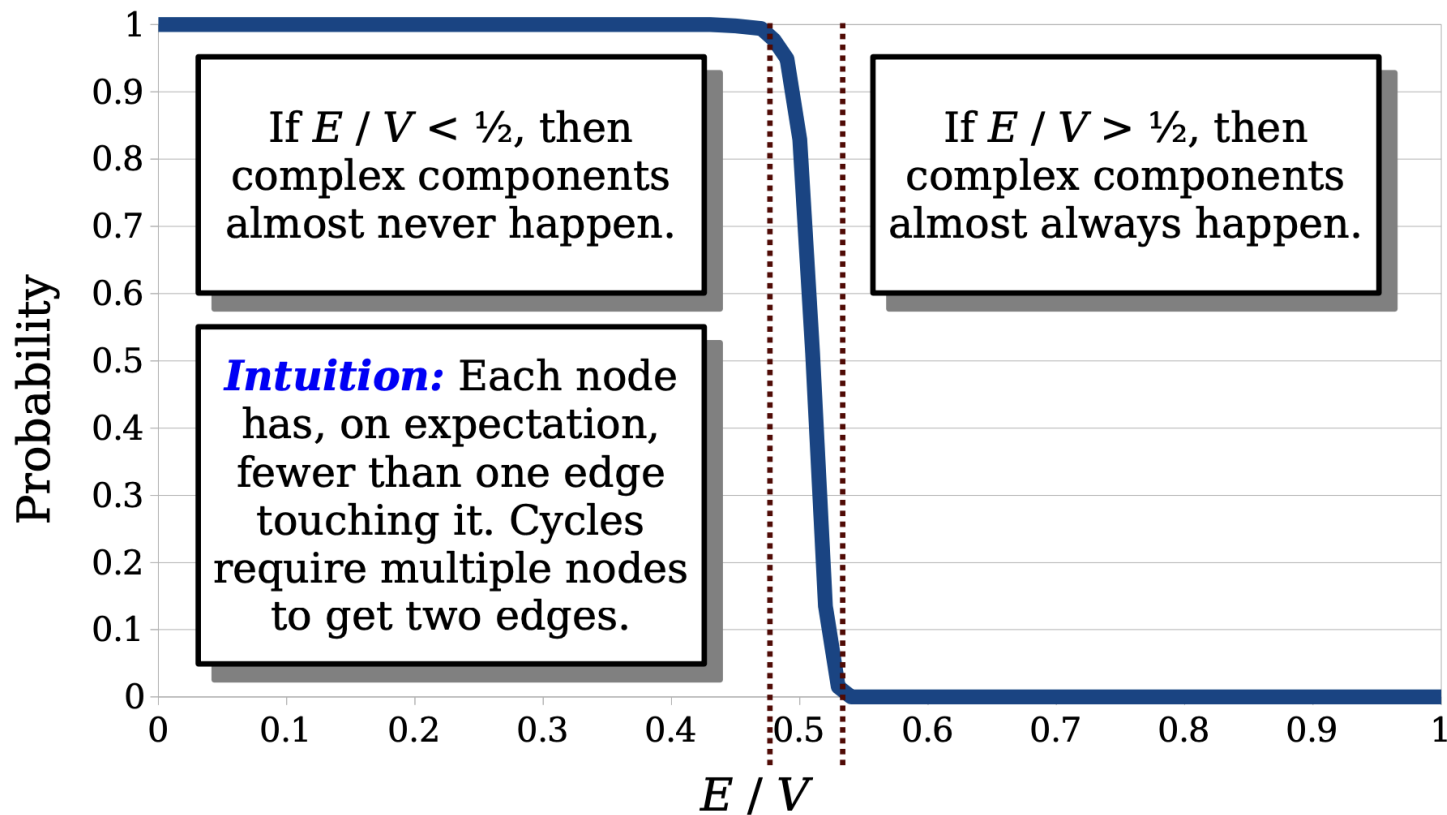
Analysis on Cuckoo Graph

- How big are the connected components in the cuckoo graph?
 - This tells us how much work we do on a successful insertion
- What is the probability that a connected component in the cuckoo graph that contains two or more cycles?
 - This lets us see how much time we should expect to spend rehashing
- A random graph model (V, E)
 - With $|V| = m$ nodes and $|E| = N$ edges
 - Each edge joins a uniformly random pair of slots, by the choice of two hash functions

Random Graph Theory (1)



Random Graph Theory (2)



Pushing Past the 1/2 Load Factor

Blocked cuckoo hashing

Each slot holds b keys. Displace only when the slot is full.

Choose b so one slot fits a data block

d -ary cuckoo hashing

Use d hash functions, so each key has d candidate slots.

Lookups probe d slots and cost more hash evaluations.

**Both relaxations
buy space
efficiency.
Blocking is
usually faster.**

	$d = 2$	$d = 3$	$d = 4$
$b = 1$ (Standard)	50.0%	91.8%	96.5%
$b = 2$	89.7%	97.5%	99.2%
$b = 4$	97.5%	99.6%	99.9%
$b = 8$	99.7%	99.9%	>99.9%

Hashing in Modern Systems

Hash indexes

PostgreSQL hash indexes, Oracle hash clusters, MySQL memory tables.

Query execution

Hash joins and hash aggregation for in-memory query processing

Hash partitioning for distributed query processing

Sketches

Bloom filters, count-min, and HyperLogLog rely on k-wise pairwise hashing families (Lectures 5& 6)

Vector search

Locality sensitive hashing for approximate nearest neighbors (Lectures 10&11)

Takeaways

1

Hash indexes trade order for one-probe equality access.

2

Extendible and linear hashing make a hash file grow without full rehashing.

3

Guarantees come from the family of hash functions, never from a single function.

4

Cuckoo hashing reach worst-case $O(1)$ lookups by bounding candidate slots.