Tree Indexing (ctd.)
and
Hash Indexing

R&G Chapter 10,11

(slides adapted from content by J.Gehrke, J.Shanmugasundaram, and/or C.Koch)

Project 1

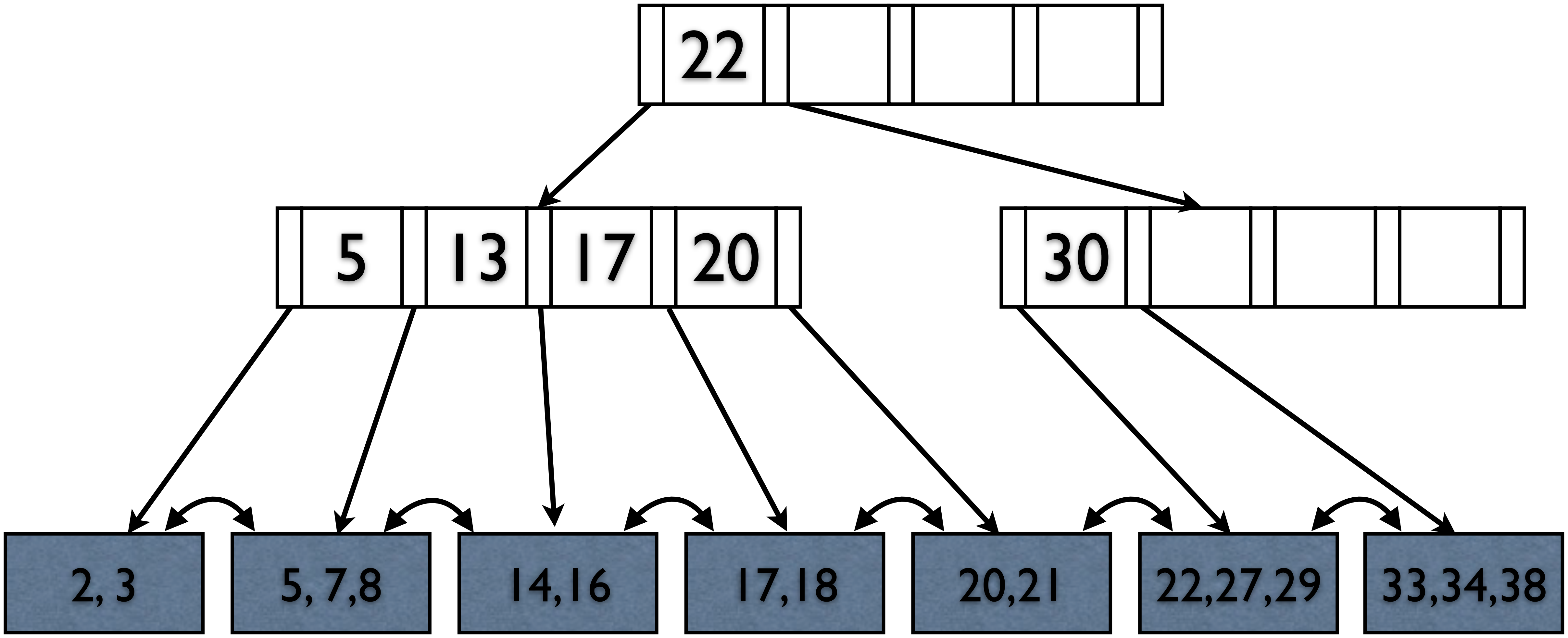
```
public static ... interpretQuery(PlanNode q)
  throws SqlException
  {
    switch(q.type){
      case AGGREGATE: {
        AggregateNode q2 = (AggregateNode)q;
      }
      ...
    }
    ...
```

Recap: Tree Indexes

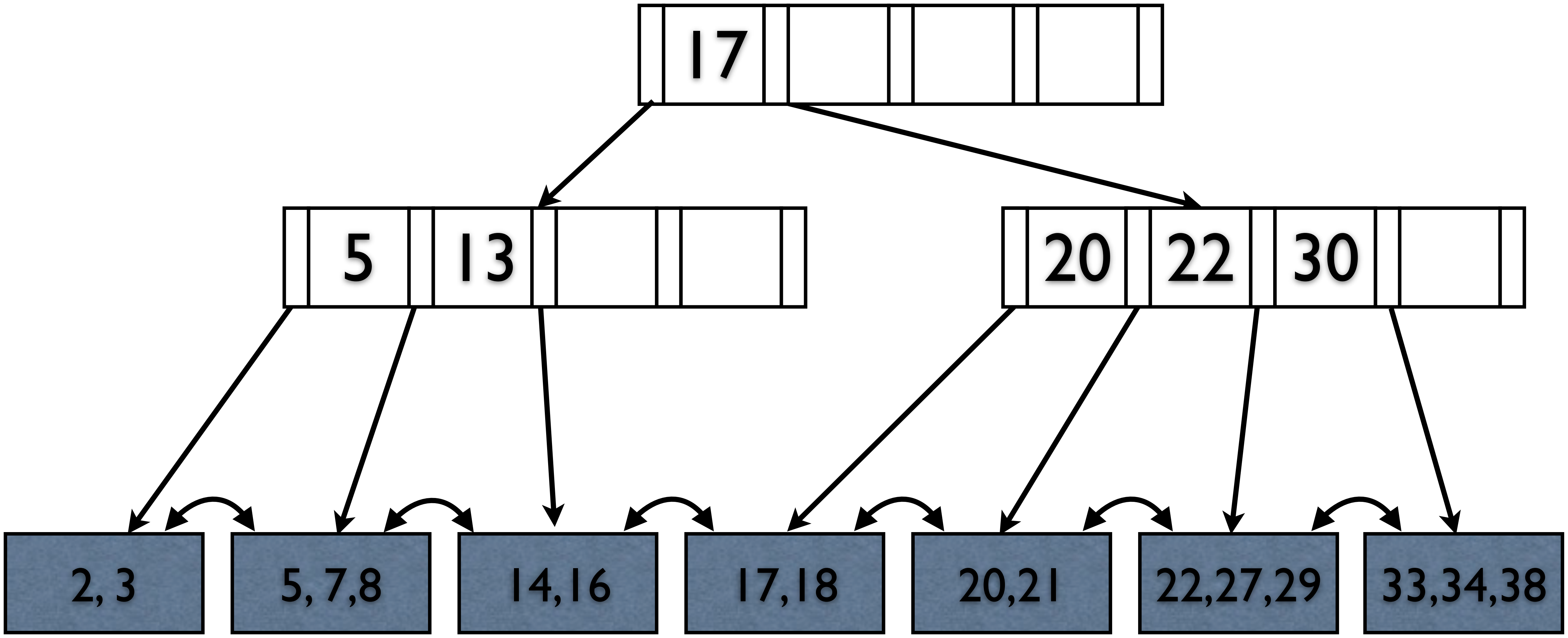
- Two types of tree-structures: ISAM, B+
  - ISAM is static, B+ is dynamic
- B+ Tree nodes split/merge as data is added/deleted.
  - Tree is kept balanced
- Tree width (fanout) is important for efficiency: Why?
  - How do we get higher fan-outs?

Higher fanouts mean shallower trees, and fewer pages loaded to find an entry. These trees are often quite shallow (depth 4-5), so even a small reduction is huge.
Page sizes are fixed, so the only way to get higher fanouts is to pack more keys/pointers into a page. This is difficult for fixed size keys, but consider variable-length strings.

Non-Leaf Redistribution

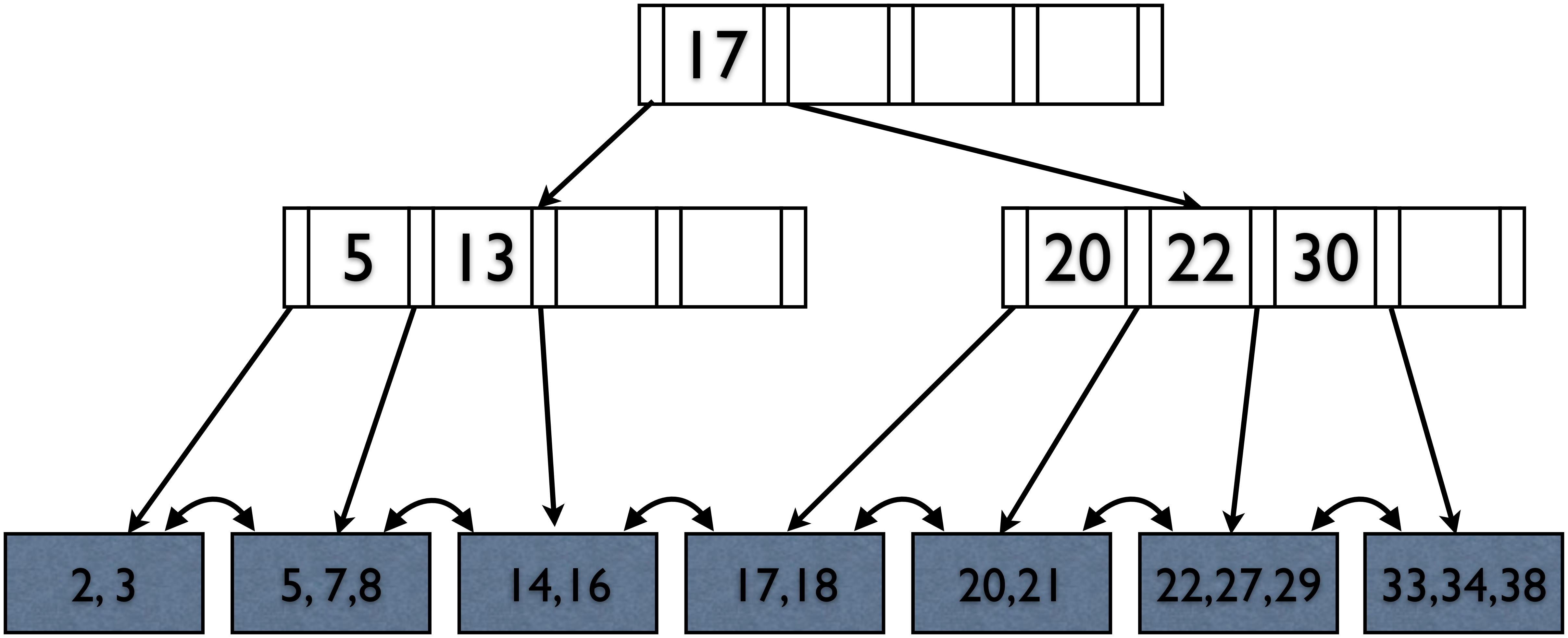


Non-Leaf Redistribution



Note that it would be sufficient to redistribute only entry 20, but we may as well even everything out.

Non-Leaf Redistribution



Intuitively, we rotate index entries 17-22 through the root

Note that it would be sufficient to redistribute only entry 20, but we may as well even everything out.


Any Questions?

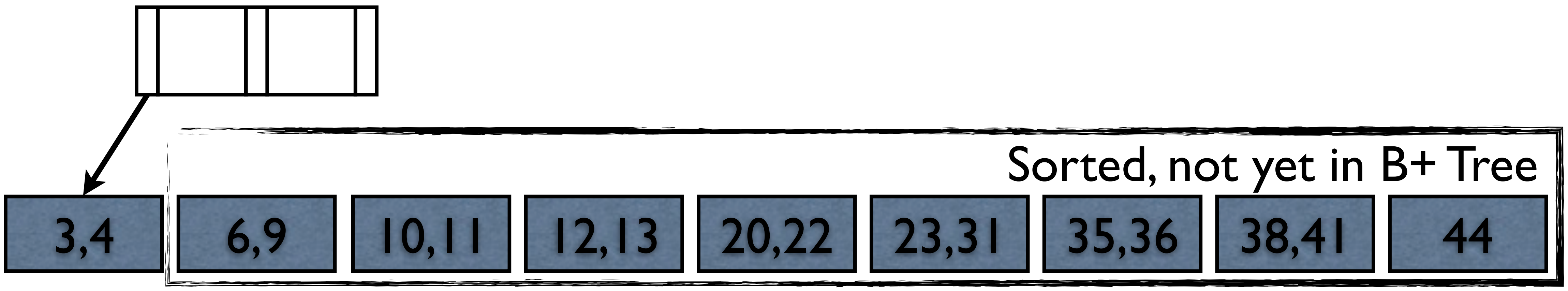
Prefix Key Compression

- We can compress key strings to a minimal prefix.
- Consider the following separator keys:
  - Dannon Yogurt, David Smith, Devarakondra Murthy
  - Abbreviate to ‘Dan’, ‘Dav’, ‘De’
- Is this strictly correct?

Not strictly correct. What happens if we’re indexing the key “Davey Jones” (which is ‘smaller’ than David Smith (e < i). We can only delete past the first differing character between the last entry in the preceding page and the first entry of the next page.

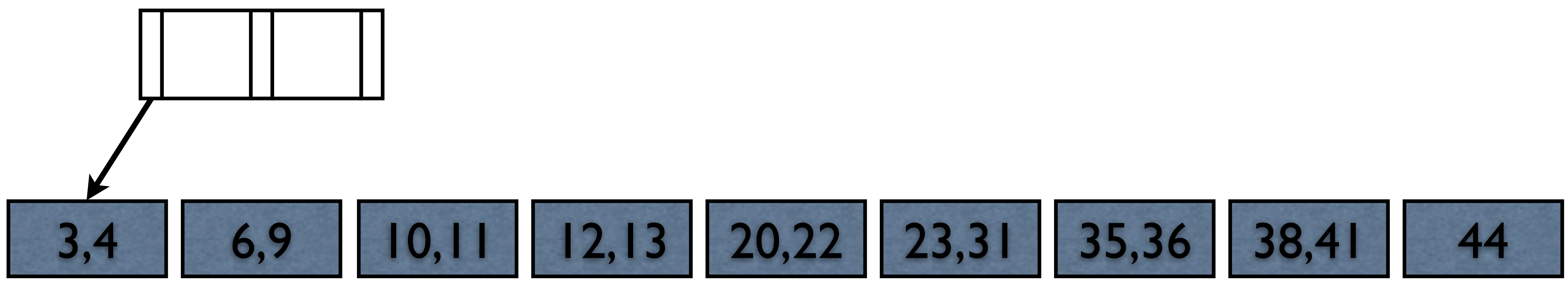
Bulk Loading of B+ Trees

- Insertions into a B+ Tree are expensive.
- What happens if we want to bulk-load a large volume of data in one go?
- Sort the data, then build up the index.
- Start with a root pointing to the first page.



Bulk Loading of B+ Trees

Keep adding fields to the rightmost index
Split when necessary

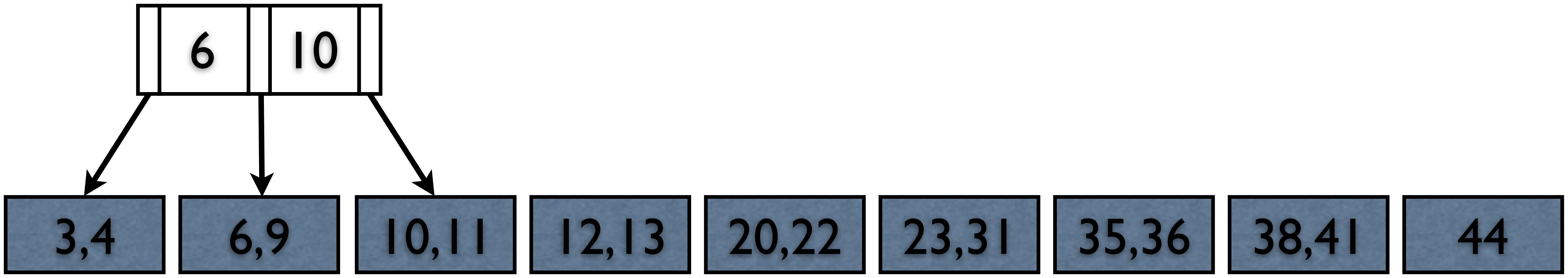


Thursday, February 7, 13

Much faster than repeated inserts. Also allows us to avoid locking overheads!

Bulk Loading of B+ Trees

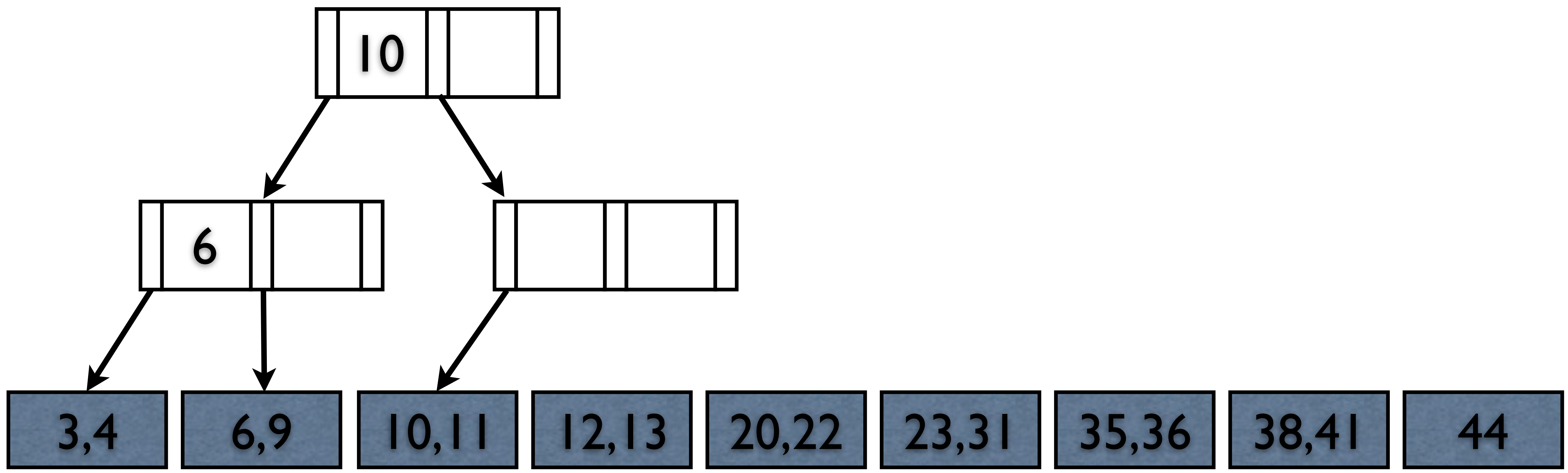
Keep adding fields to the rightmost index
Split when necessary



Much faster than repeated inserts. Also allows us to avoid locking overheads!

Bulk Loading of B+ Trees

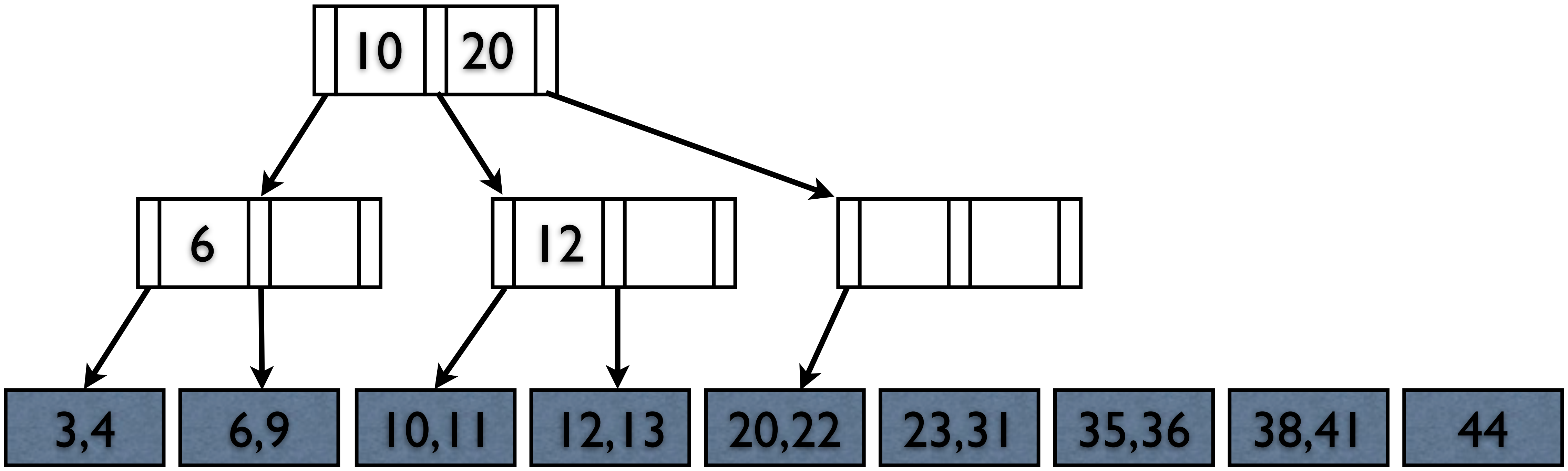
Keep adding fields to the rightmost index
Split when necessary



Much faster than repeated inserts. Also allows us to avoid locking overheads!

Bulk Loading of B+ Trees

Keep adding fields to the rightmost index
Split when necessary

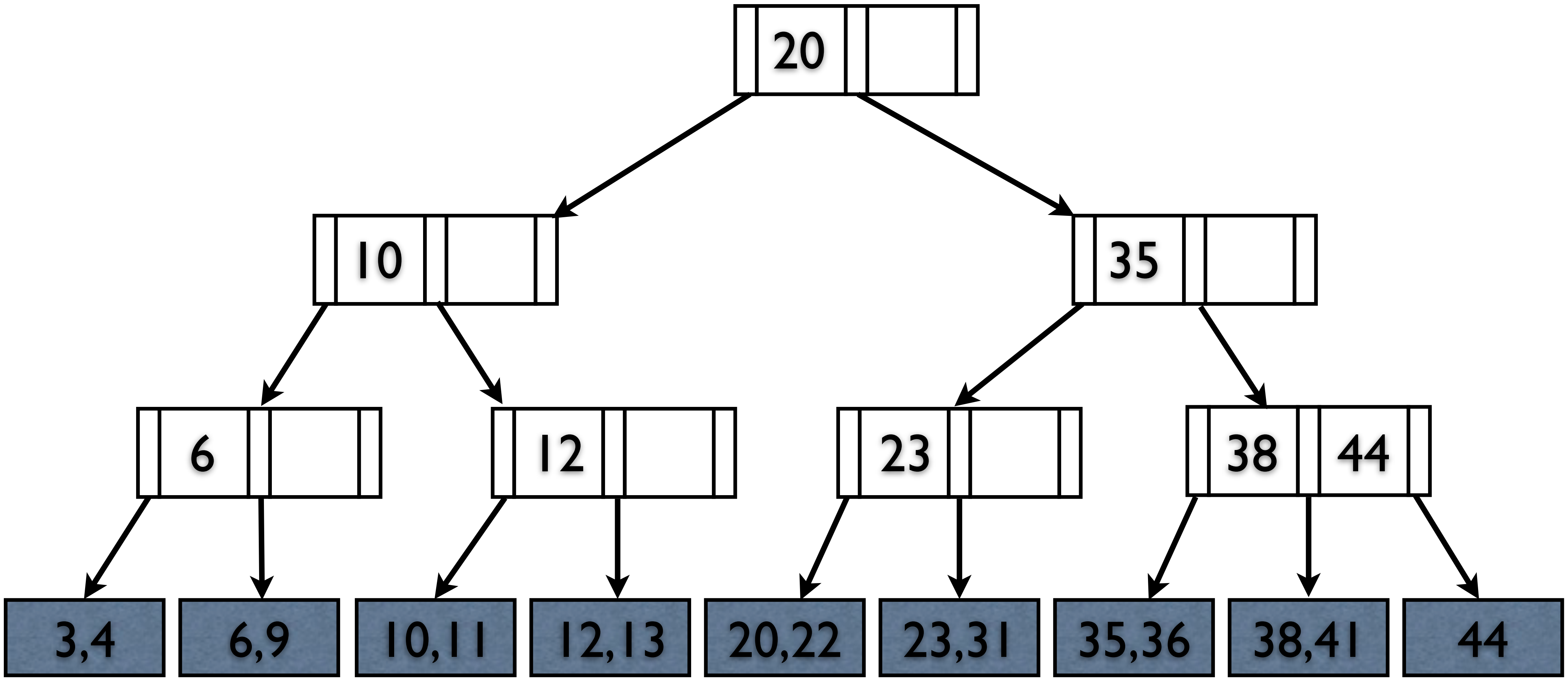


Thursday, February 7, 13

Much faster than repeated inserts. Also allows us to avoid locking overheads!

Bulk Loading of B+ Trees

Keep adding fields to the rightmost index
Split when necessary



Much faster than repeated inserts. Also allows us to avoid locking overheads!

Bulk Loading of B+ Trees

- Batch operation is more efficient under concurrency control
- Fewer IOs during build. How many?
- Leaves will be stored sequentially. How?
- Can easily control the ‘fill factor’

This operation will require O(N*log(N)) IOs for the sort. We only ever modify the rightmost entry at each level of the index, so as long as we can keep log(N) pages in memory (depth of the index), we can do the entire index construction in N reads + N/fanout writes (index over N pages contains N pages)
Leaves will be stored sequentially (initially) because we can pre-allocate a sequential block as the last stage of the sort. This is great, because it means our linked-list scans will involve sequential reads (we can store which pages are still sequential in the catalog)


Any Questions?

A Note on ‘Order’

- B+Tree guarantee: d < # entries < 2d
- In practice, we use % of page filled
  - Can typically keep more entries in index pages
  - Variable sized records make d hard to predict
  - Multiple records with the same value of the search key can lead to variable sized data entries

B+ Trees in Practice

- Typical order: 100; Fill factor 67%
  - Avg Fanout: 133
- Typical Capacities
  - Depth 4: 133^4 = 312,900,700 records
  - Depth 3: 133^3 = 2,352,637 records
- Top levels often fit in buffer pool

Root: 1 page; 8KB
Level 2: 133 pages: 1MB
Level 3: 17689 pages: 133 MB

Hash-Based Indexes

Hash-based Indexes

- As with trees: request a key k and get record(s) or record id(s) with k.
- Hash-based indexes support equality lookups
  - … in constant time (vs log(n) for tree)
  - … but don't support range lookups
- Static vs Dynamic Hashing
  - Tradeoffs similar to ISAM vs B+Tree

Hash Functions

- A hash function is a function that maps a large data value to a small fixed-size value
  - Typically is deterministic & pseudorandom
- Used in Checksums, Hash Tables, Partitioning, Bloom Filters, Caching, Cryptography, Password Storage, …
- Examples: MD5, SHA1, SHA2
  - MD5() part of OpenSSL (on most OSX / Linux / Unix)
- Can map h(k) to range [0,N) with h(k) % N (modulus)

Generic hash function implementations are provided with most major programming language libraries (C's md5, Java's Object.hash, OCaml's Hashtbl).
If you're interested in learning more, two good resources are wikipedia and Press et al.'s "Numerical Recipes"

Static Hashing

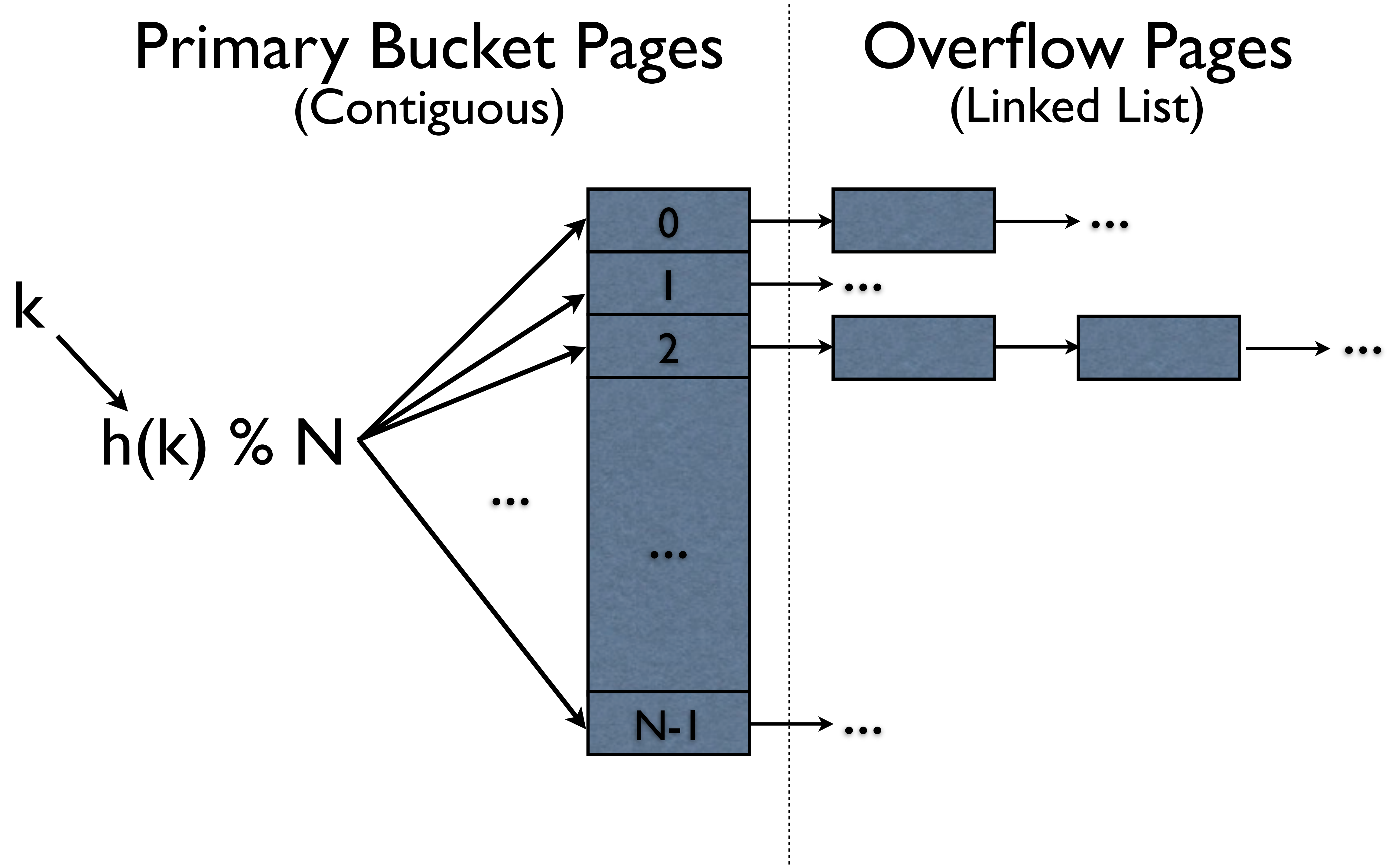


Image copyright: Paramount Pictures

Thursday, February 7, 13

How do you pick the number of hash buckets?

Static Hashing

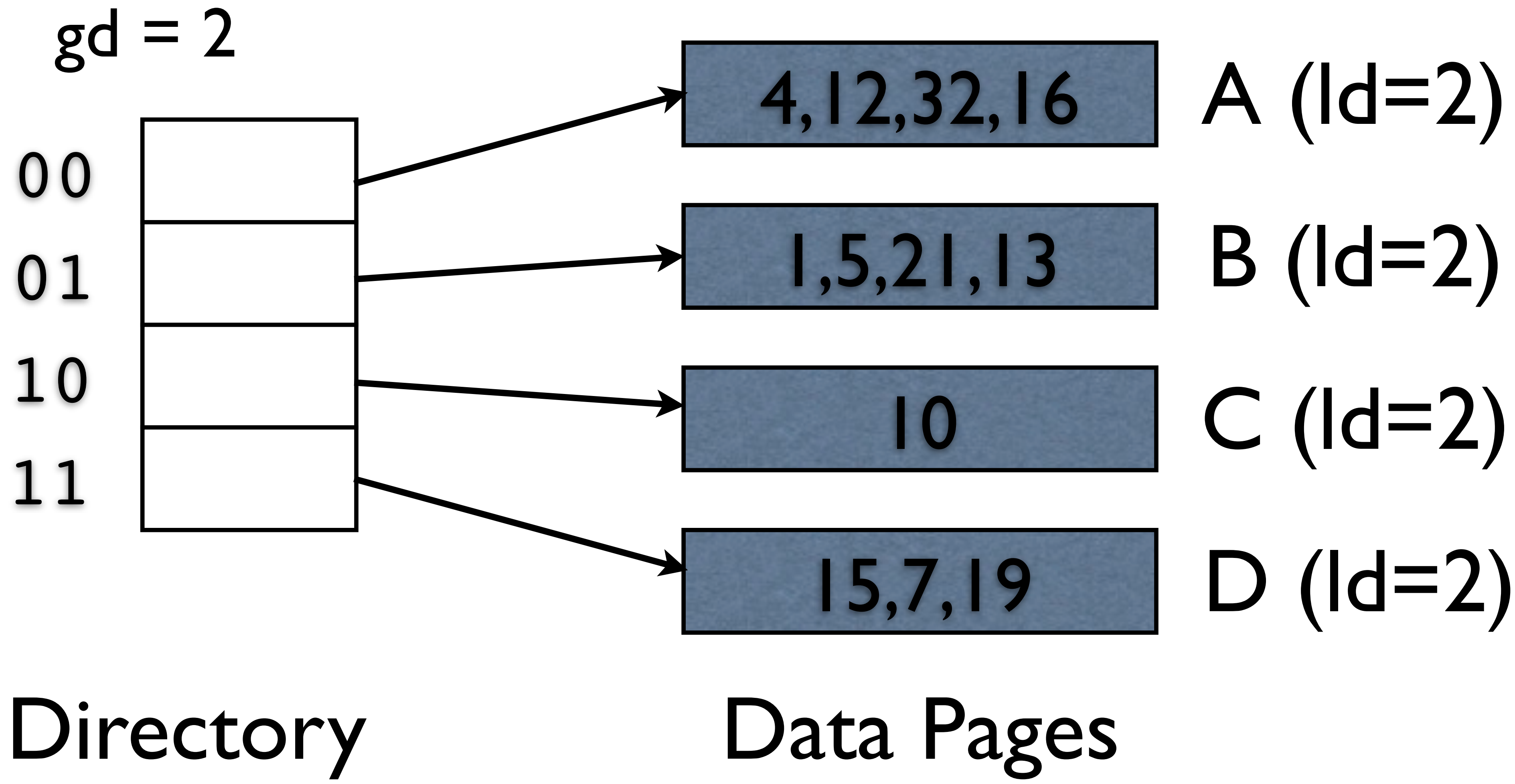
- Buckets contain data entries.
- Hash fn maps the search key field of records to one of a finite number of buckets (% N)
- N chosen when the index is created
  - Too small N: Long overflow chains
  - Too big N: Wasted space/Poor IO
- Dynamic Solutions: Extendible and Linear Hashing

Extendible Hashing

- **Situation:** A bucket becomes full
  - Solution: Double the number of buckets!
  - Expensive! (N reads, 2N writes)
- **Idea:** Add one level of indirection
  - A directory of pointers to (noncontiguous) bucket pages.
  - Doubling just the directory is much cheaper.
  - Can we double only the directory?

The hash function (and directory) must be changed (from a range of [0,N) to a range of [0,2N)) without affecting data values in buckets not affected by the split.

Extendible Hashing

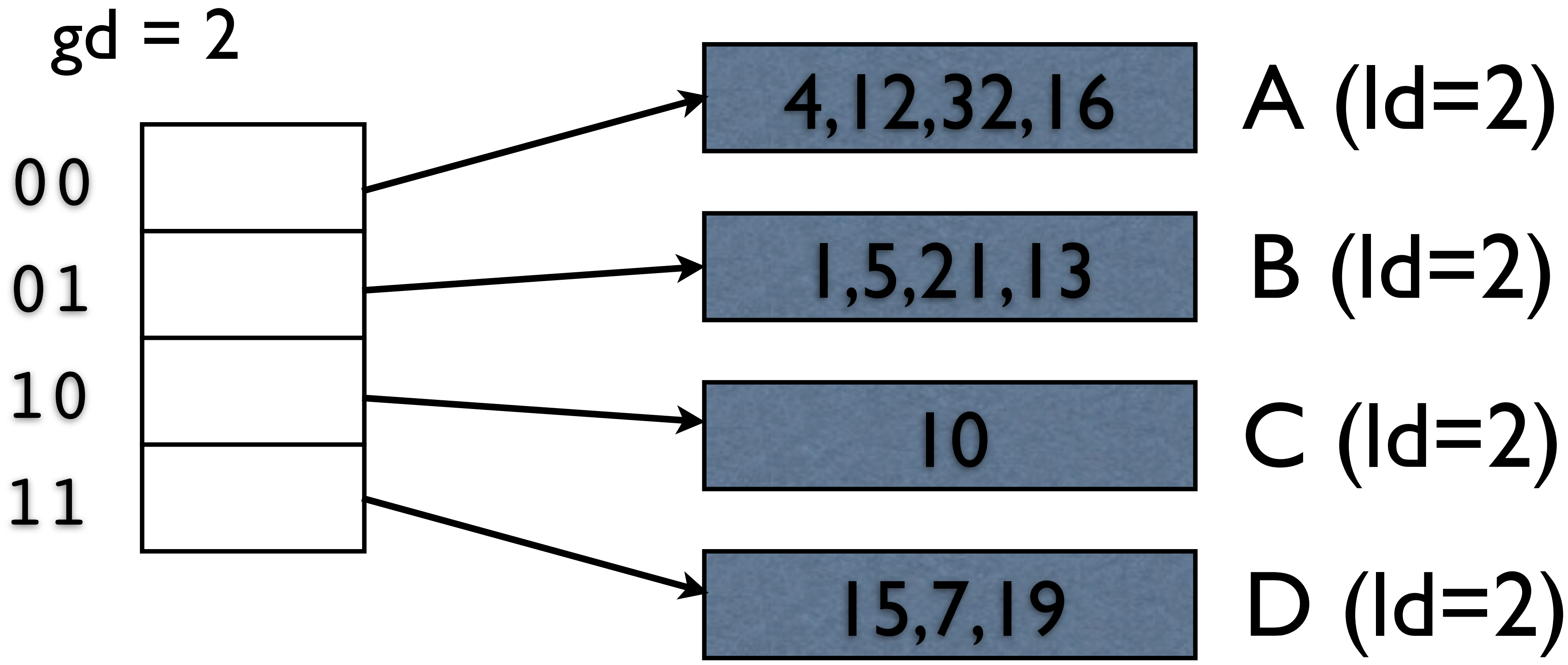


The directory and data pages have an associated "depth" (global/local)

To look up a value use the last **gd** bits of the key's hash value as an index into the dir

Global depth = log_2(directory size)
Lookup example: Assume h(12) = 0100; We have gd = 2, so we take the last 2 bits (00) and find the corresponding data page (A).

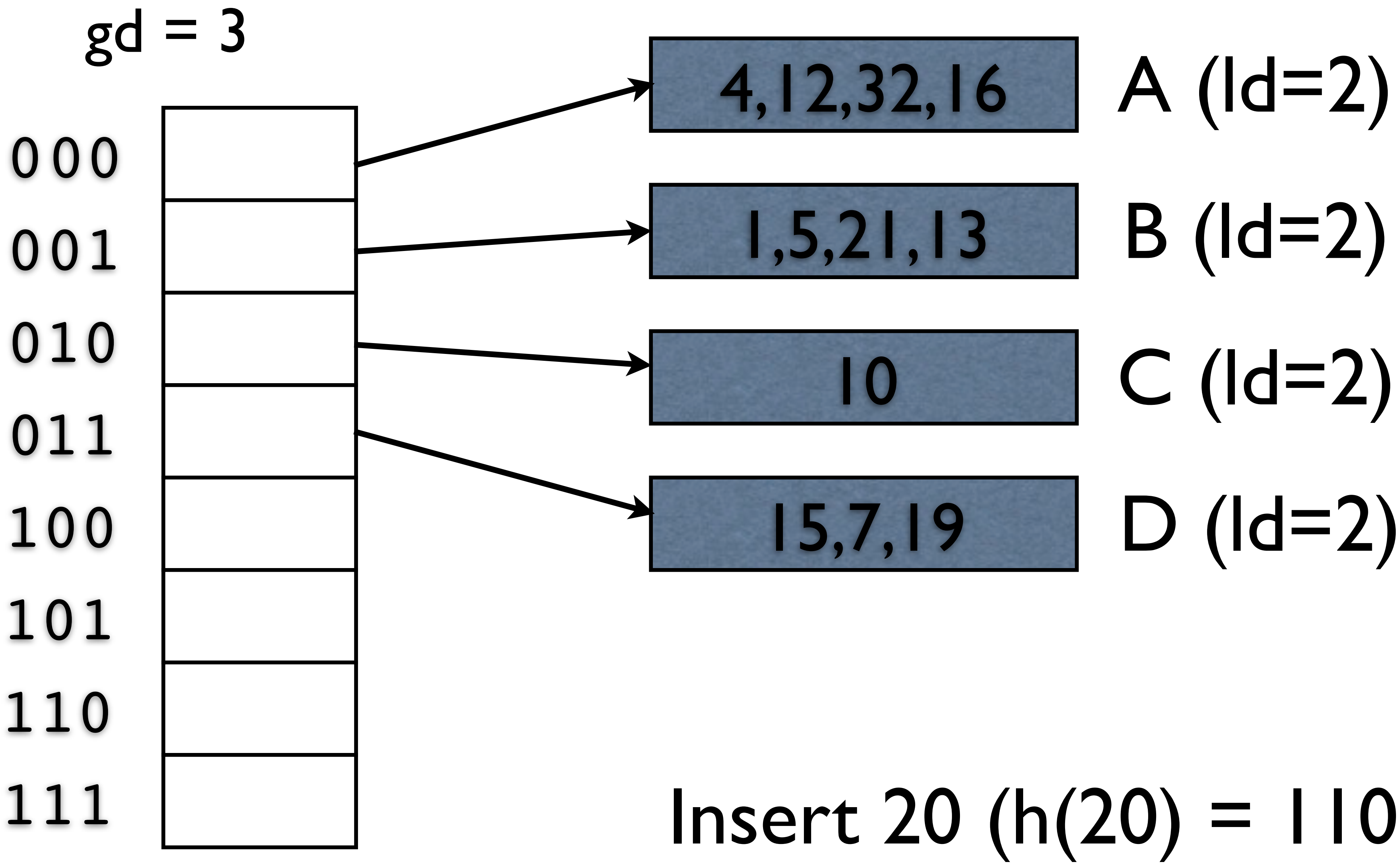
Extendible Hashing



Insert 20 (h(20) = 1100)

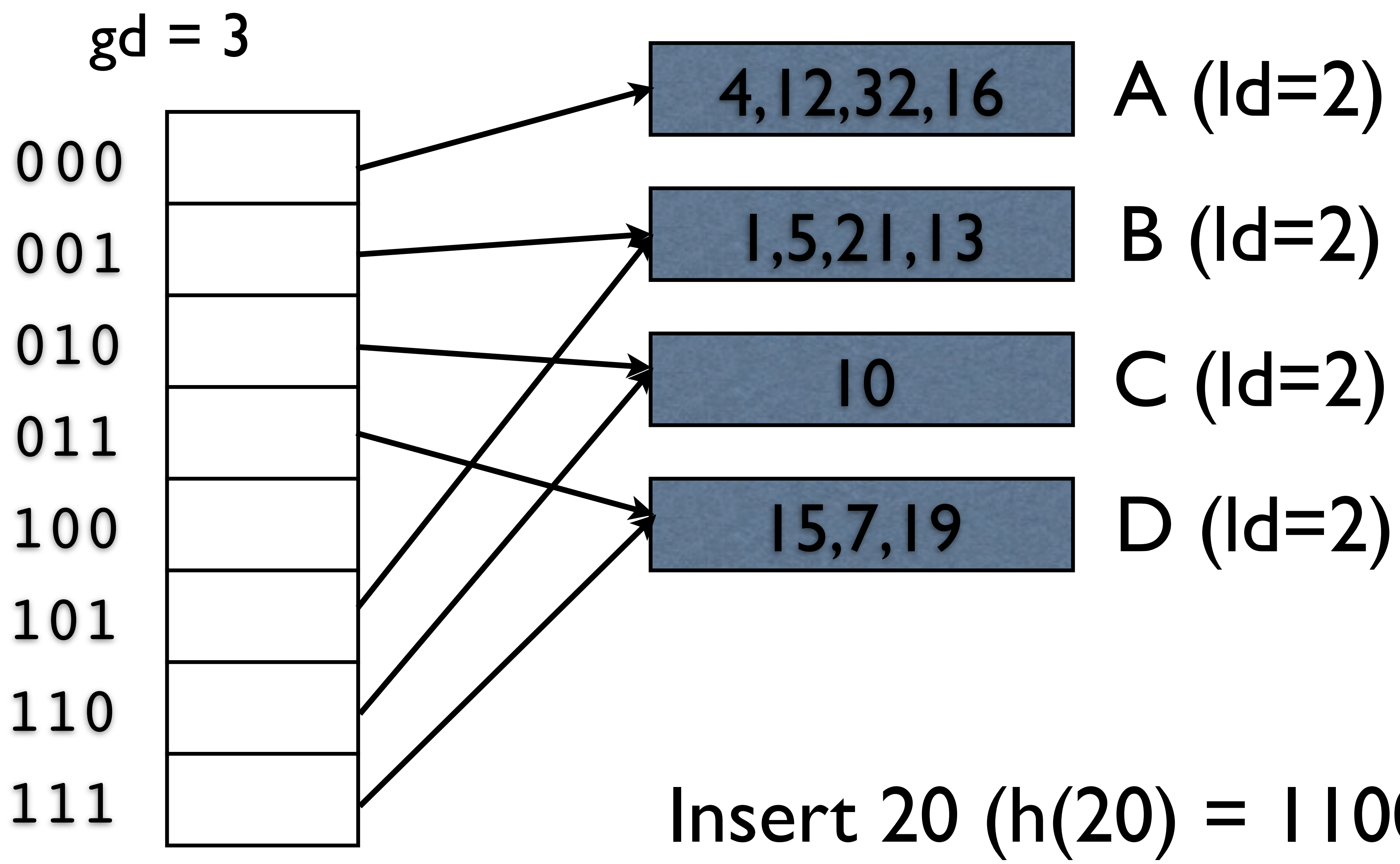
(Need to Split Bucket A)

Extendible Hashing



Insert 20 (h(20) = 1100)
(Need to Split Bucket A)

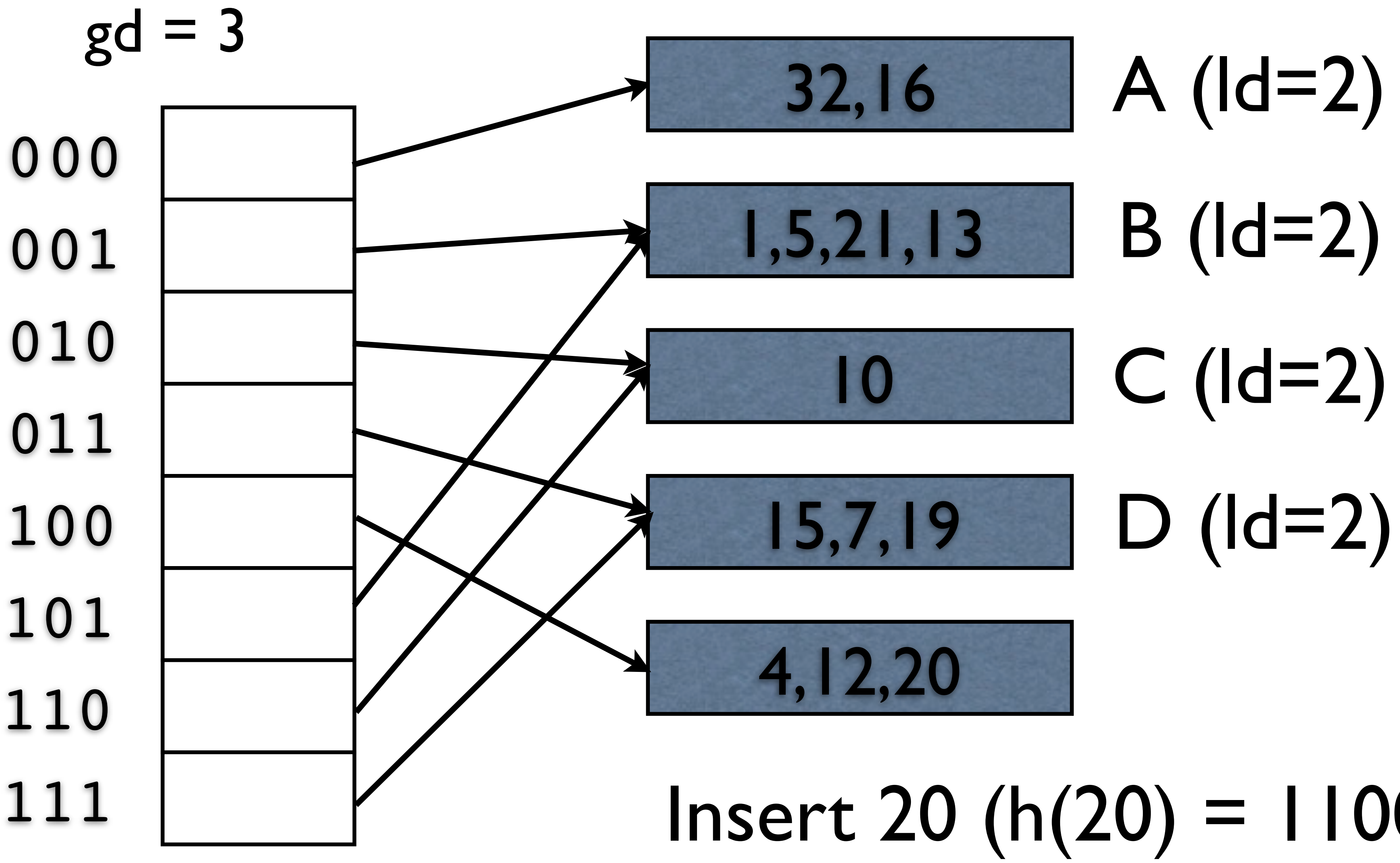
Extendible Hashing



Dir entries not being split point to the same bucket

Insert 20 (h(20) = 1100)
(Need to Split Bucket A)

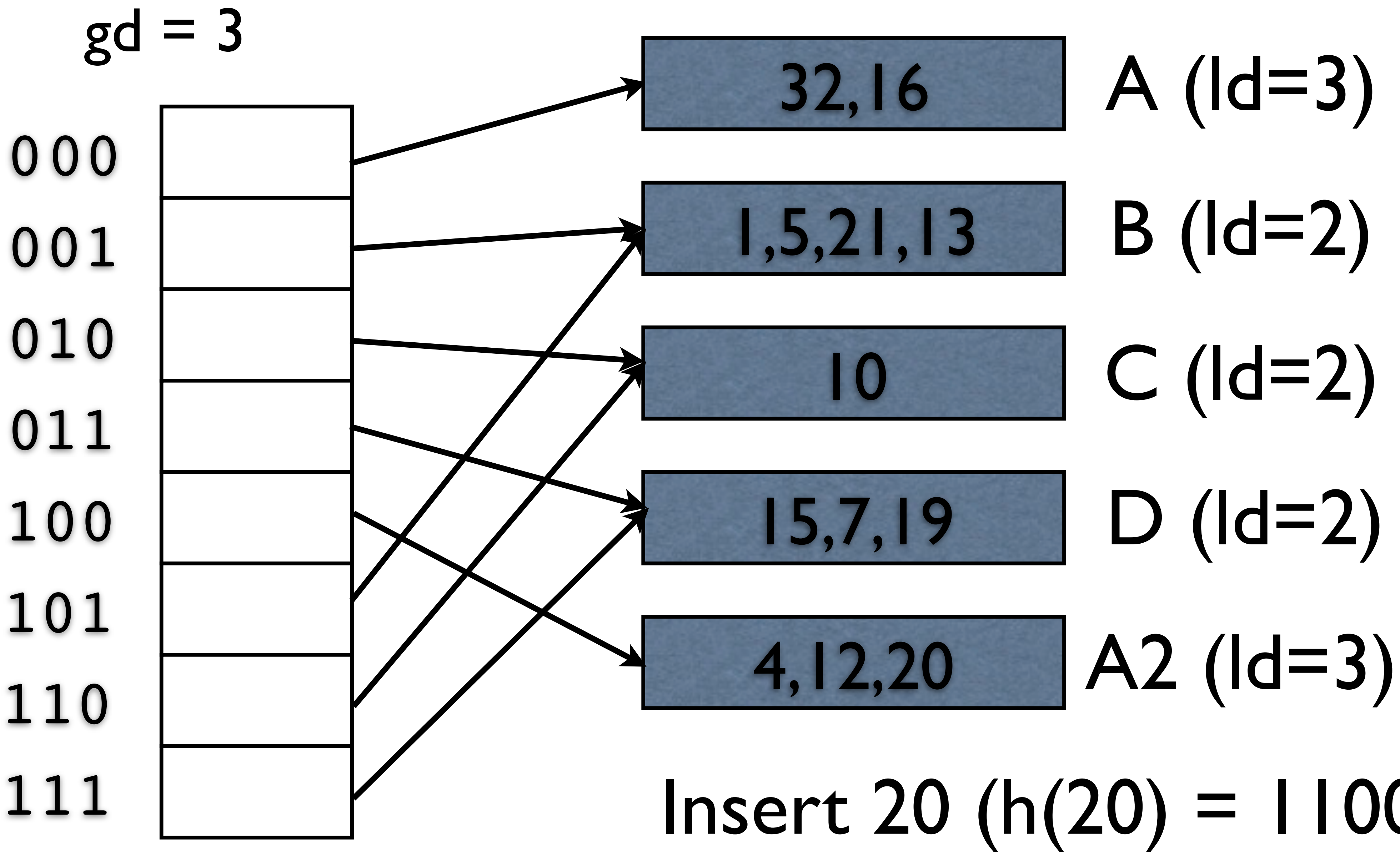
Extendible Hashing



Dir entries not being split point to the same bucket

Insert 20 (h(20) = 1100)
(Need to Split Bucket A)

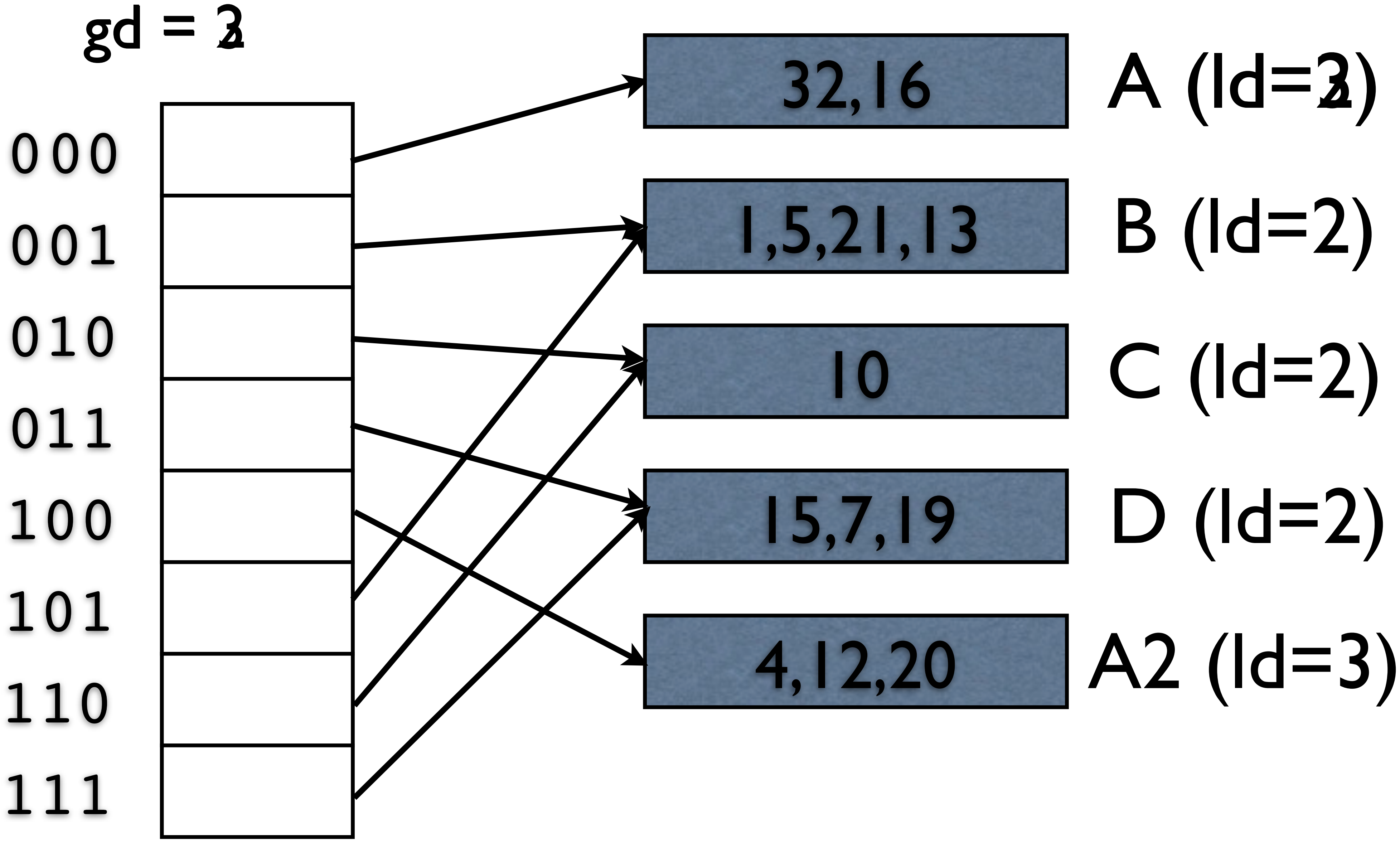
Extendible Hashing



Dir entries not being split point to the same bucket

Insert 20 (h(20) = 1100)
(Need to Split Bucket A)

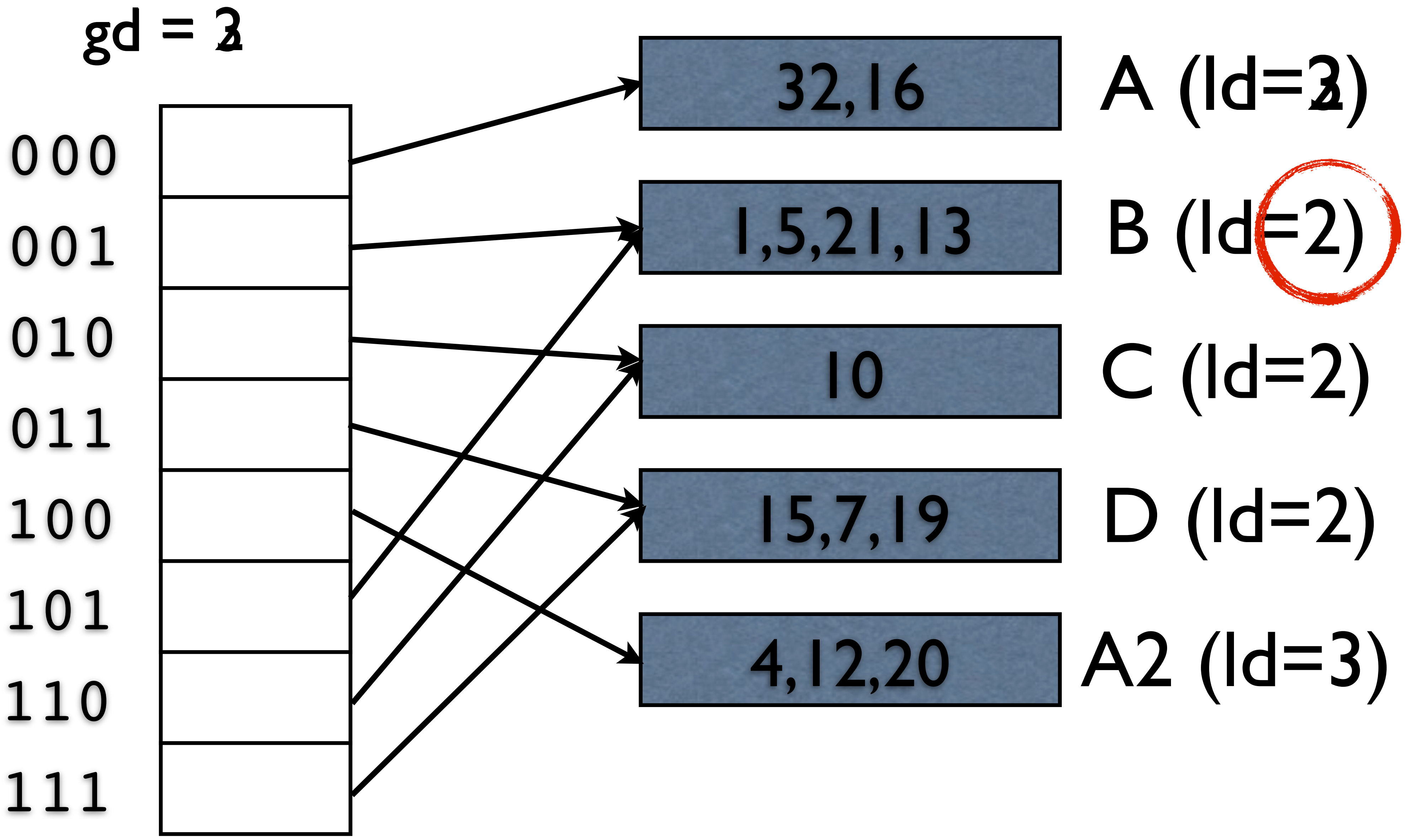
Extendible Hashing



Insert 31 (h(31) = 1001)
(Need to Split Bucket B)

As long as ld = gd, there's more than one directory pointer pointing at that page. Increment ld for the page being split, and just update the existing directory pointers appropriately.

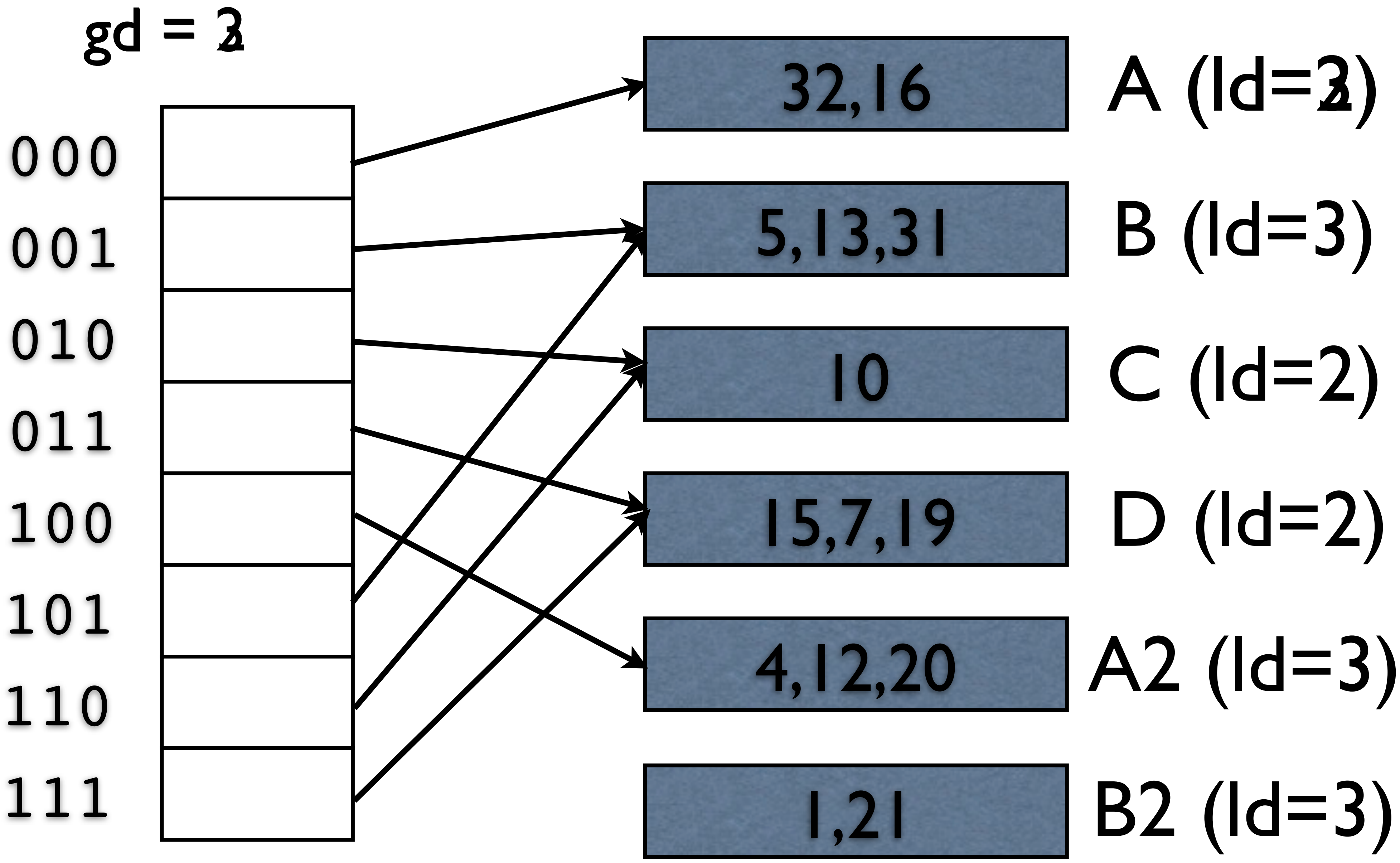
Extendible Hashing



Insert 31 (h(31) = 1001)
(Need to Split Bucket B)

As long as ld = gd, there’s more than one directory pointer pointing at that page. Increment ld for the page being split, and just update the existing directory pointers appropriately.

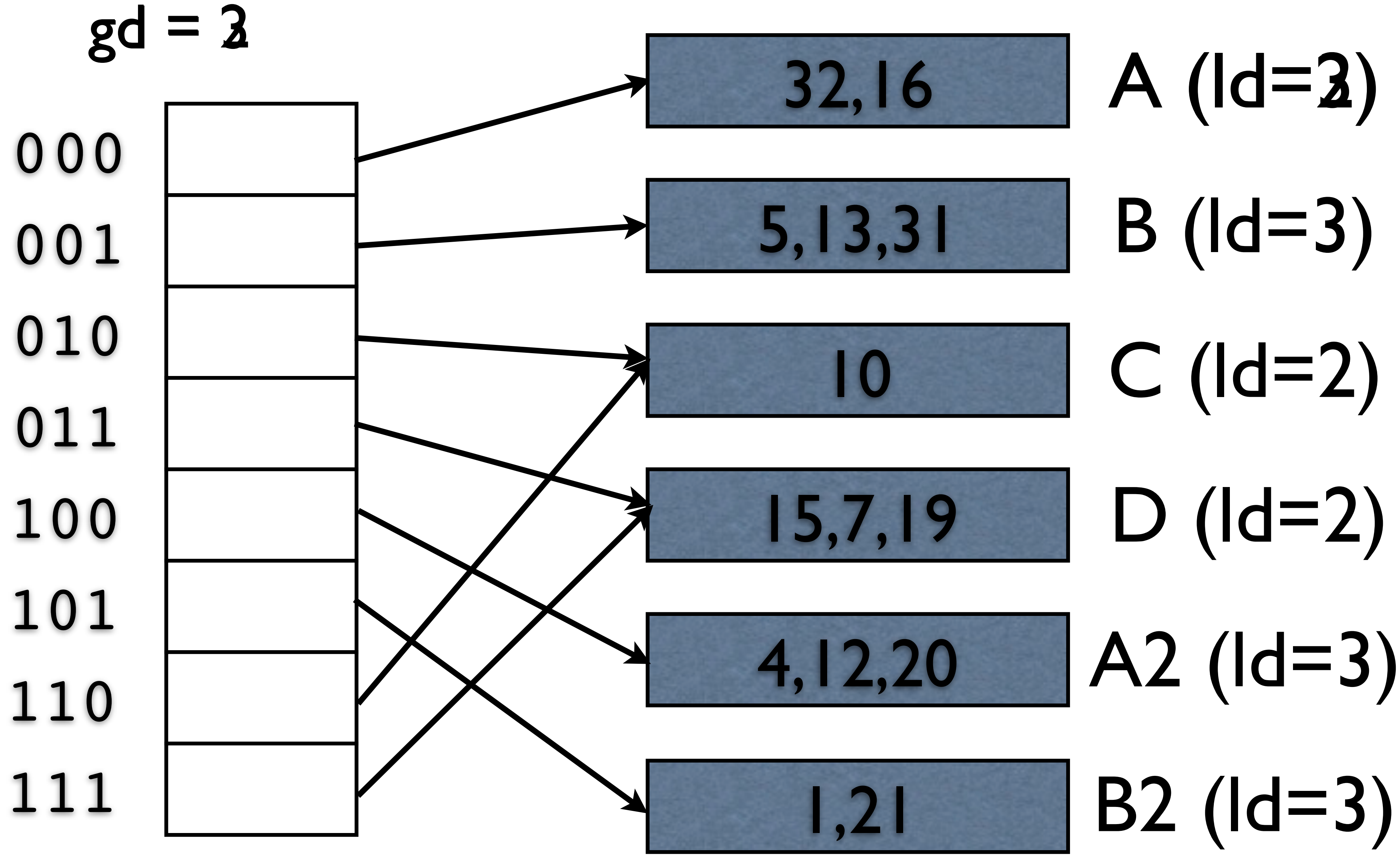
Extendible Hashing



Insert 31 (h(31) = 1001)
(Need to Split Bucket B)

As long as ld = gd, there's more than one directory pointer pointing at that page. Increment ld for the page being split, and just update the existing directory pointers appropriately.

Extendible Hashing



Don't need to double dir when splitting bucket w/ ld < gd

Insert 31 (h(31) = 1001)
(Need to Split Bucket B)

As long as ld = gd, there's more than one directory pointer pointing at that page. Increment ld for the page being split, and just update the existing directory pointers appropriately.

Extendible Hashing

- Global depth of directory
  - **Upper bound** on # of bits required to determine the bucket of an entry.
- Local depth of a bucket
  - **Exact** # of bits required to determine if an entry belongs in this bucket.
- Why use least significant bits (vs MSB)?

If LSBs are used, new directory entries appear contiguously (see ex on slide 16) at the end. We can do a bulk copy of all entries. If MSBs are used, new entries appear interleaved with old entries, and copying is more computationally expensive (and has higher IO costs)

Extendible Hashing

- If the entire directory fits in memory, any equality search can be answered in one disk access. (otherwise two)
  - Is this true even if the directory spans multiple pages?
  - 100 MB file, 100 B/rec = 1m records over 4k pages.
    - Minimum of 25k directory entries.
    - Hash table still likely to be < 1M

Extendible Hashing

- Hashing Issues:
  - Need a uniform distribution of hash values.
    - Even a true random function will not provide this
  - What could happen if multiple keys have the same hash value? (A hash 'collision')
- Deletions
  - Deleting the last entry in a bucket allows it to be merged with its 'split image'.
  - Can potentially halve directory if this happens.


Any Questions?

Linear Hashing

- A directory page adds 1 page lookup overhead.
- Can we do similar splits without indirection?
- Linear Hashing based on similar principle.
  - Start with the last *n* bits of each hash fn.
  - When you decide to split, start using *n+1* bits.
- **Key difference**: Split incrementally
  - Part of the hash table uses *n* bits, rest uses *n+1*
  - Each *round* increase *n* by one (1 round = 1 full split)

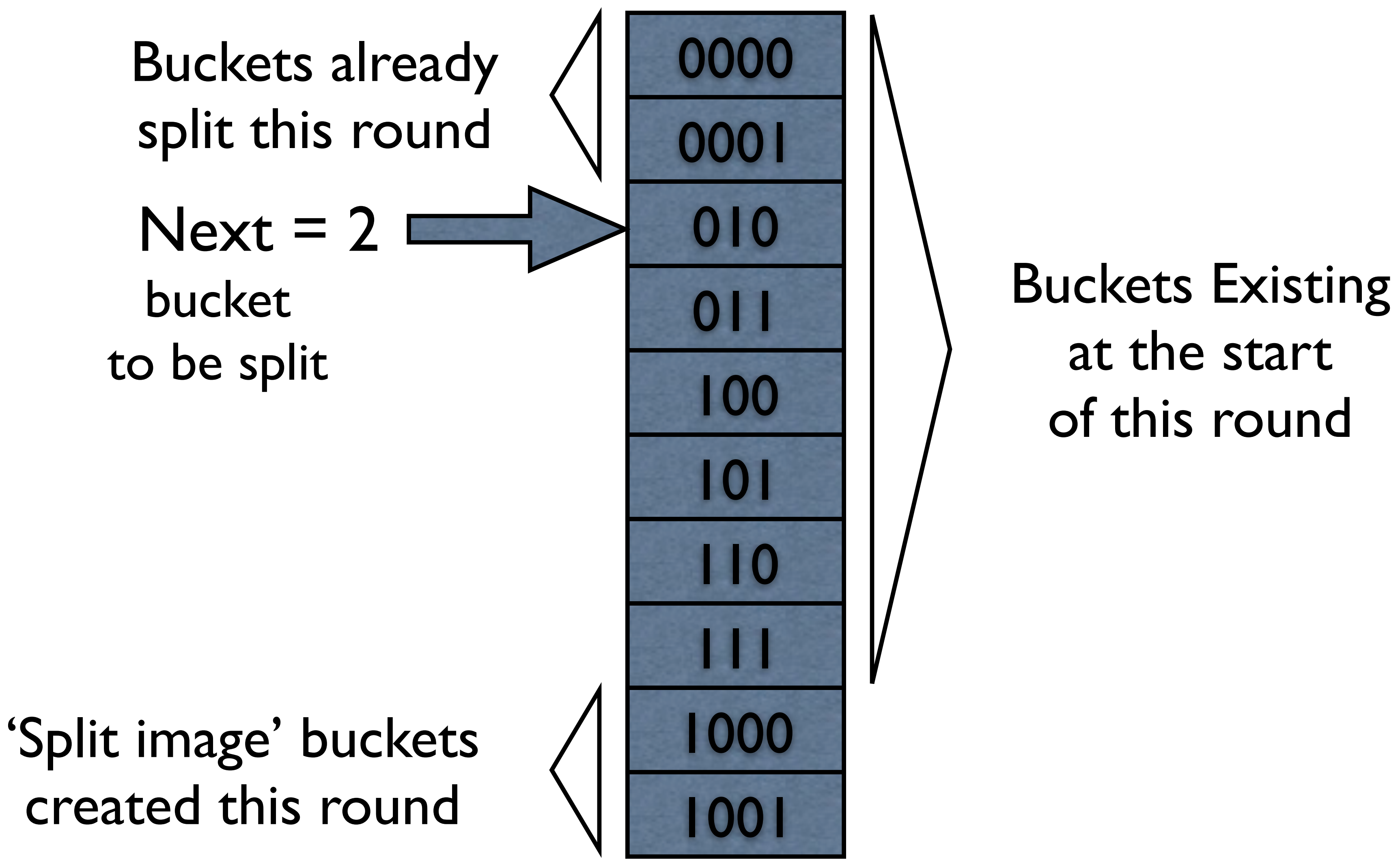
We can generalize the splitting idea a little bit: We're taking one hash function h(k), and defining a new function: h'(k,n) = h(k) % 2^n (2 to the nth). Another way to look at this is that we're defining a family of hash functions h'1(k) = h(k) % 2, h'2(k) = h(k) % 3, h'4(k) = h(k) % 8, …

Any family of hash functions that satisfies the copy on split property can be switched in for this one

- That is, we can swap in any family as long as h'n(k) = h'(n+1)(k) % 2^n

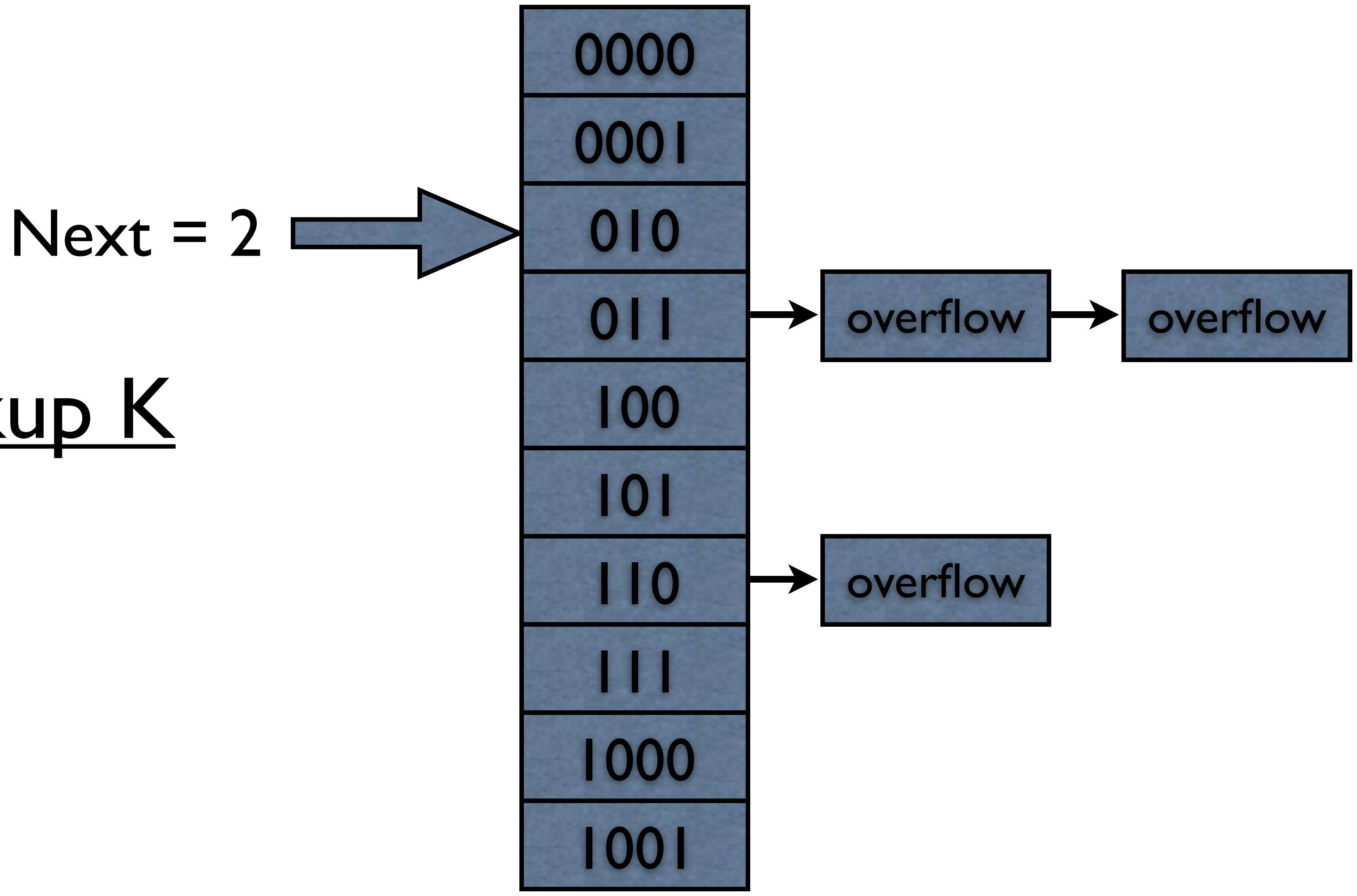
Linear Hashing

Level = 3 (2^3 = 8 Entries)



Linear Hashing:Lookups

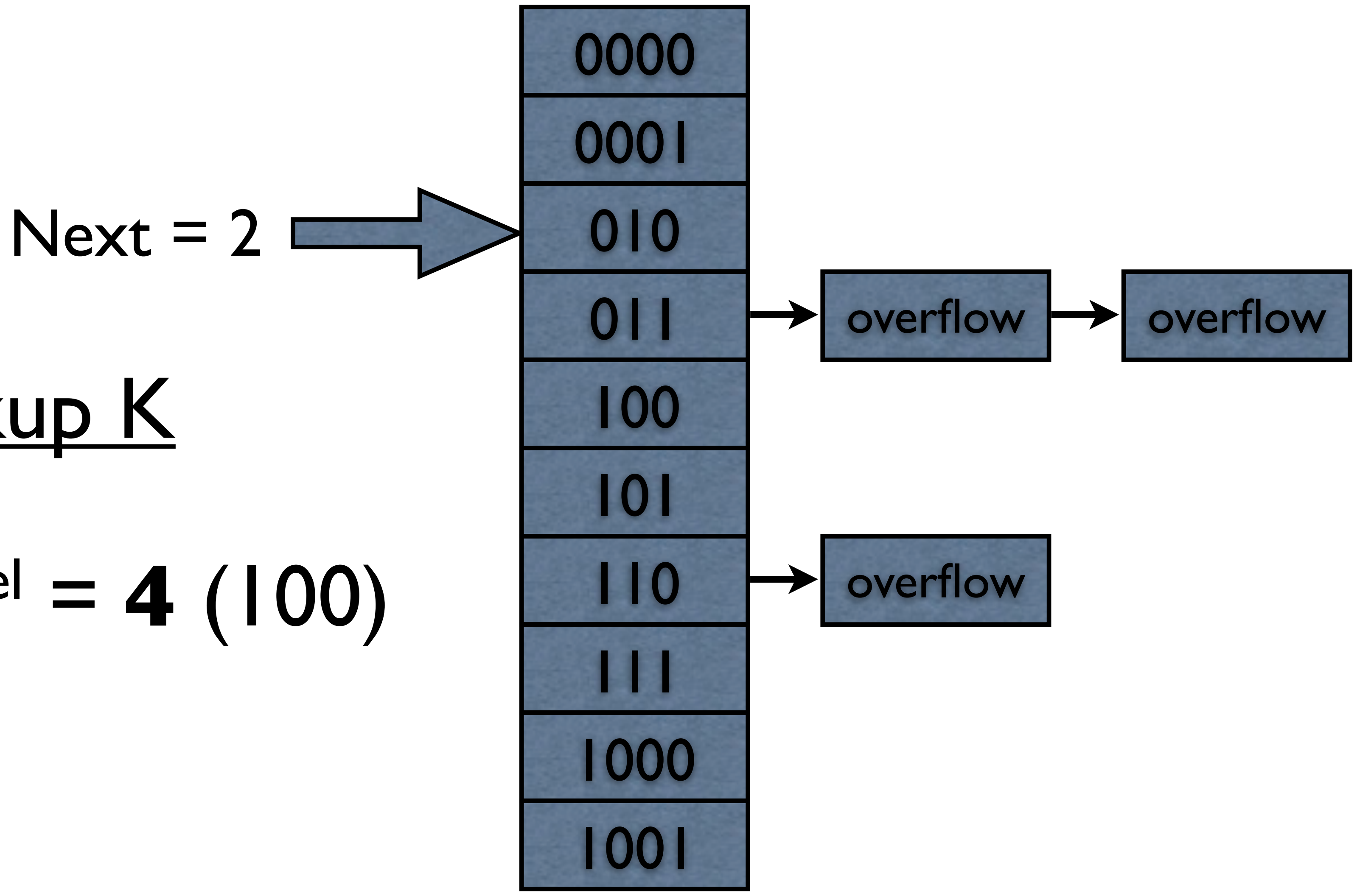
Level = 3 (2^3 = 8 Entries)



Lookup K

Linear Hashing:Lookups

Level = 3 (2^3 = 8 Entries)



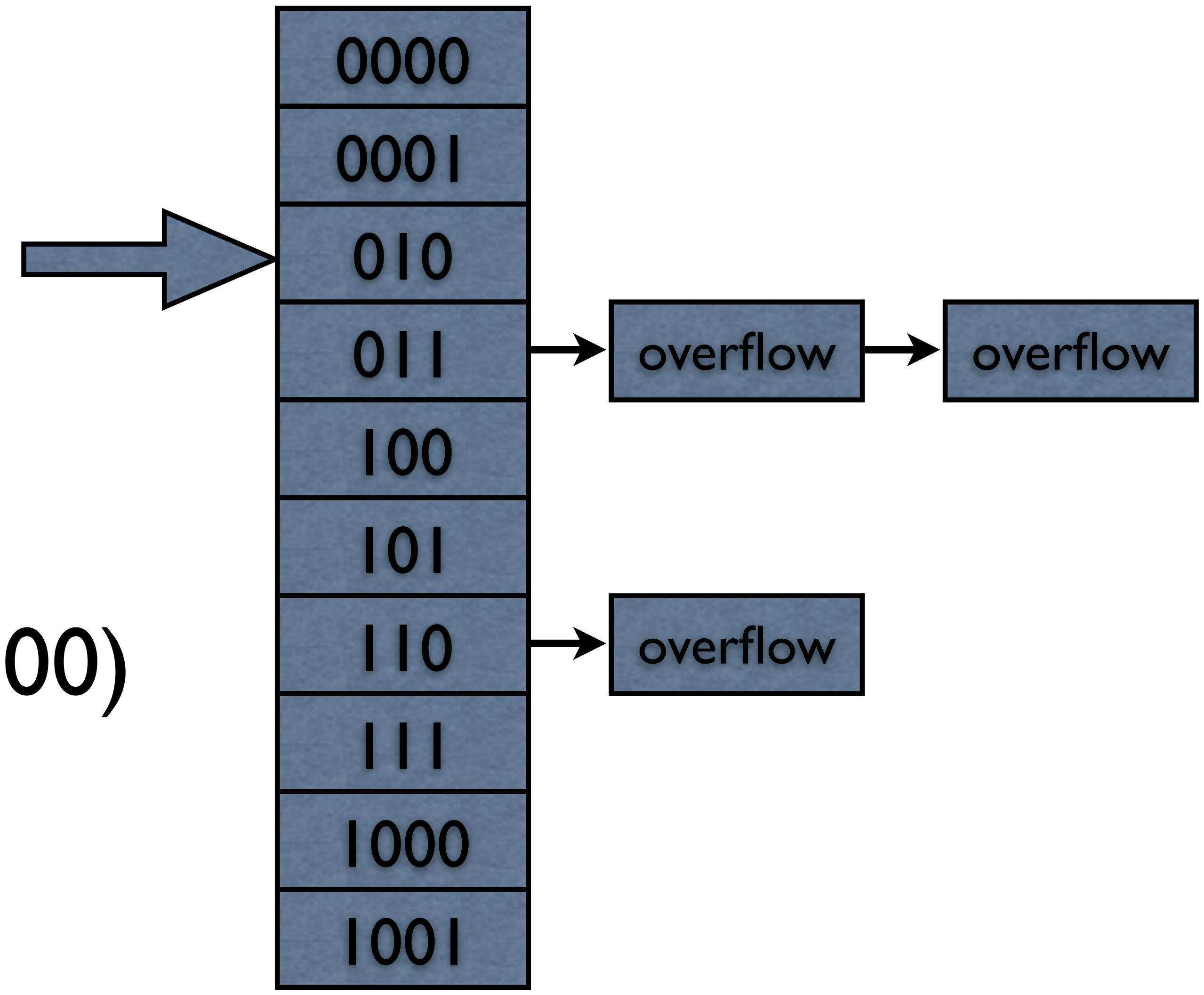
Lookup K

h(k) % 2^{Level} = **4** (100)

Linear Hashing:Lookups

Level = 3 (2^3 = 8 Entries)

Next = 2



<u>Lookup K</u>

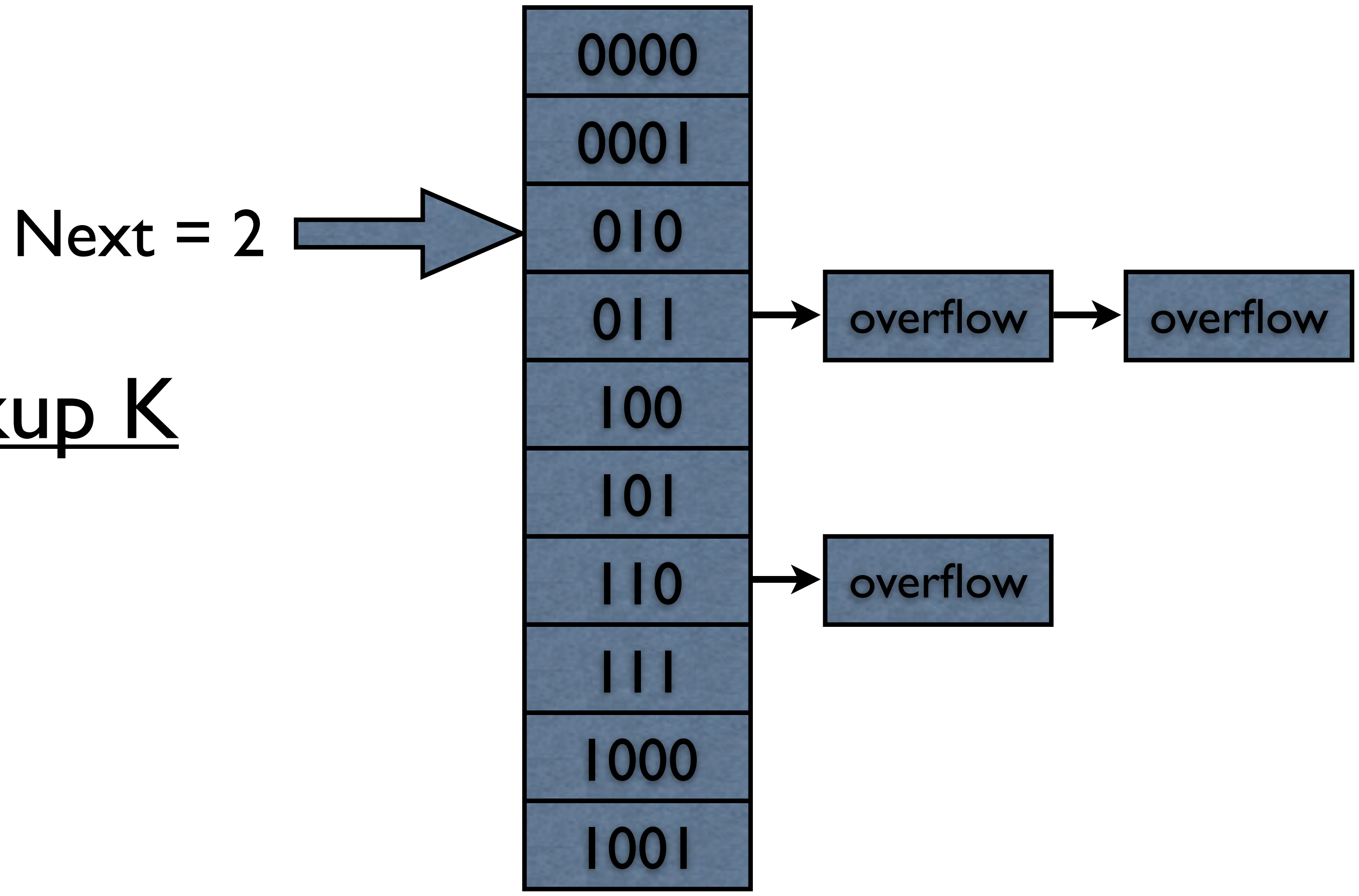
$h(k) \% 2^{Level} = \mathbf{4}$ (100)

Next $\leq$ **4**

Use entry 4

Linear Hashing:Lookups

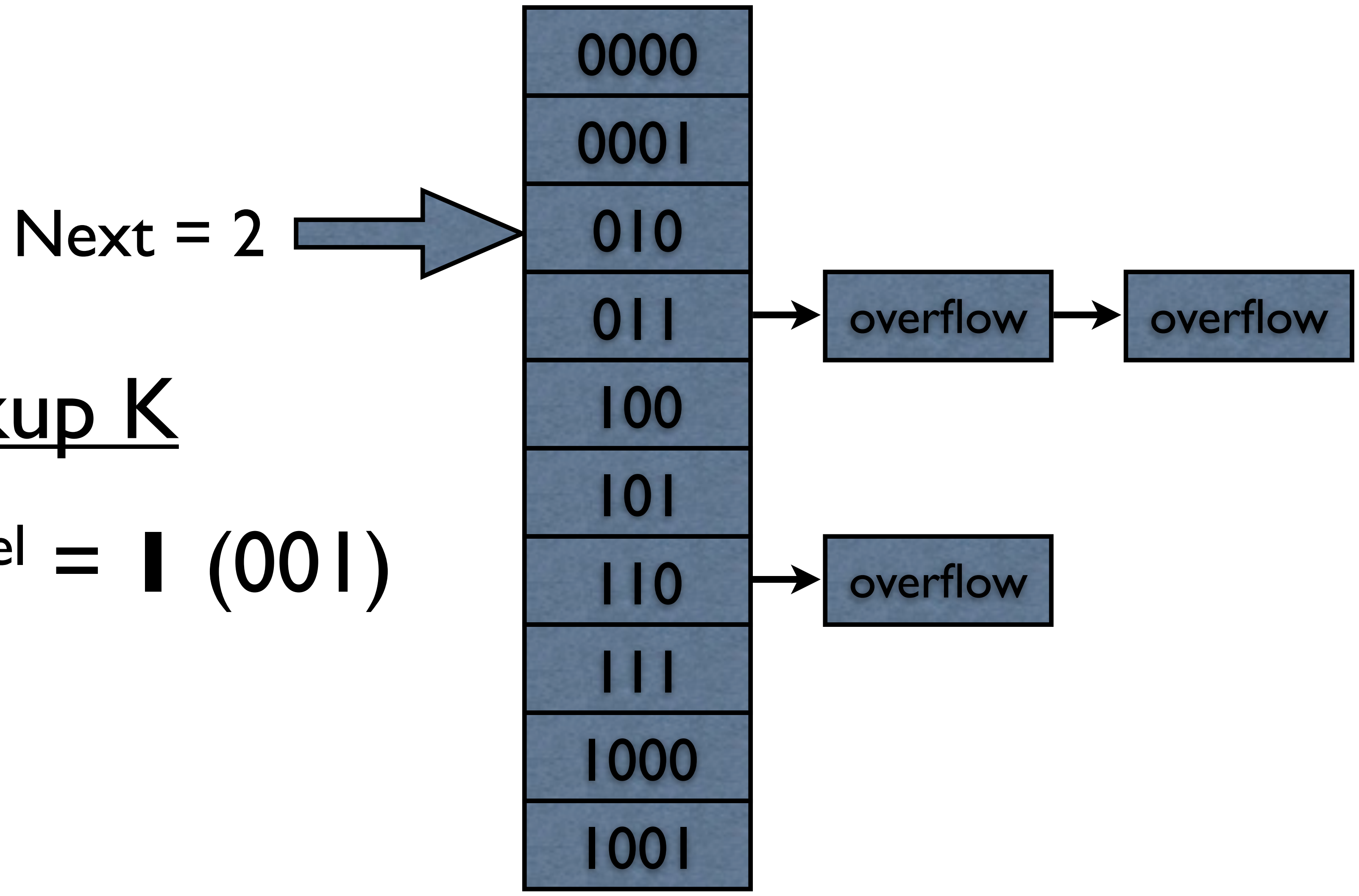
Level = 3 (2^3 = 8 Entries)



Lookup K

Linear Hashing:Lookups

Level = 3 (2^3 = 8 Entries)

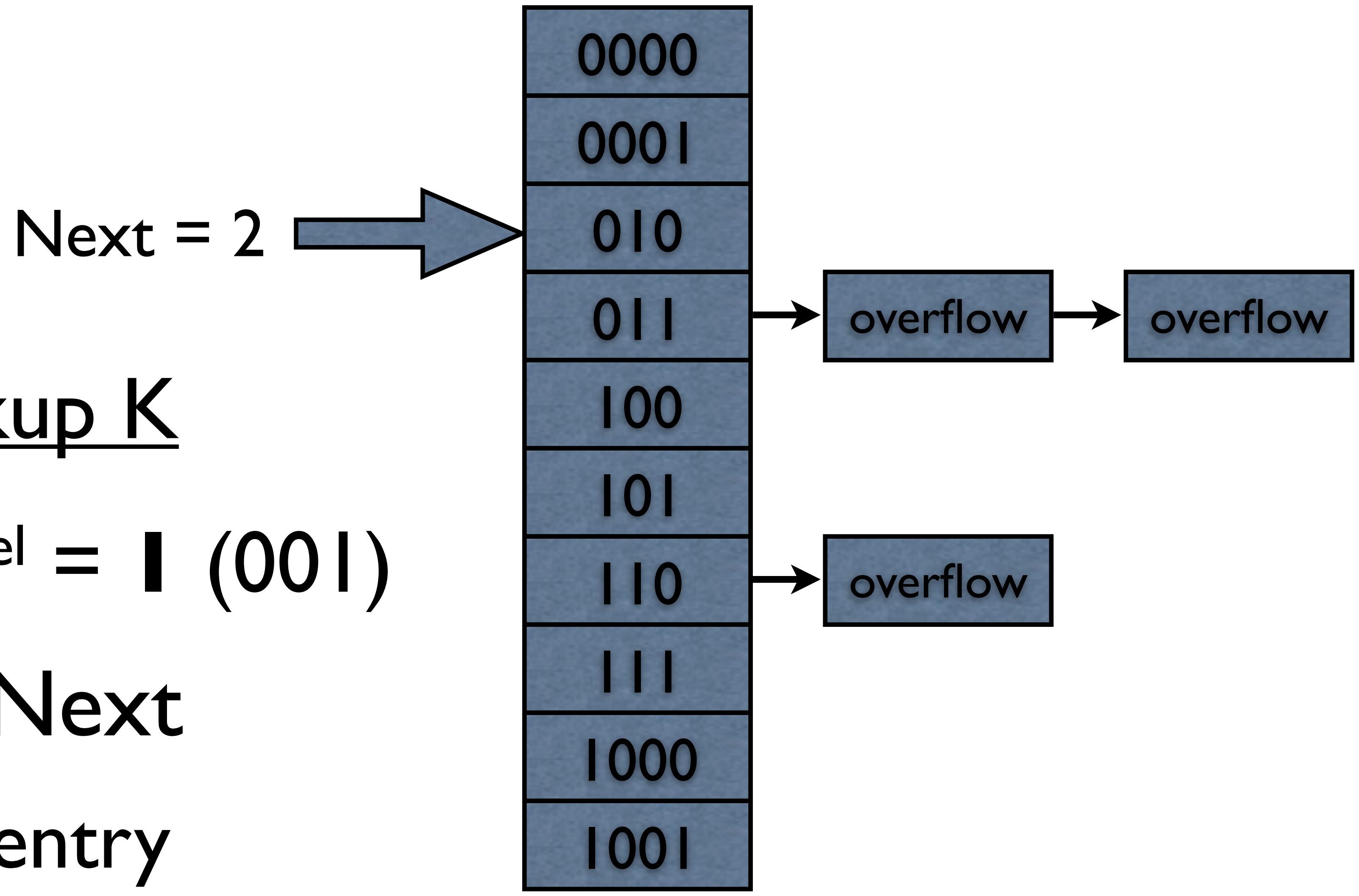


Lookup K

$h(k) \% 2^{Level} = 1$ (001)

Linear Hashing:Lookups

Level = 3 (2^3 = 8 Entries)



Lookup K

$h(k) \% 2^{Level} = 1$ (001)

1 < Next

Use entry

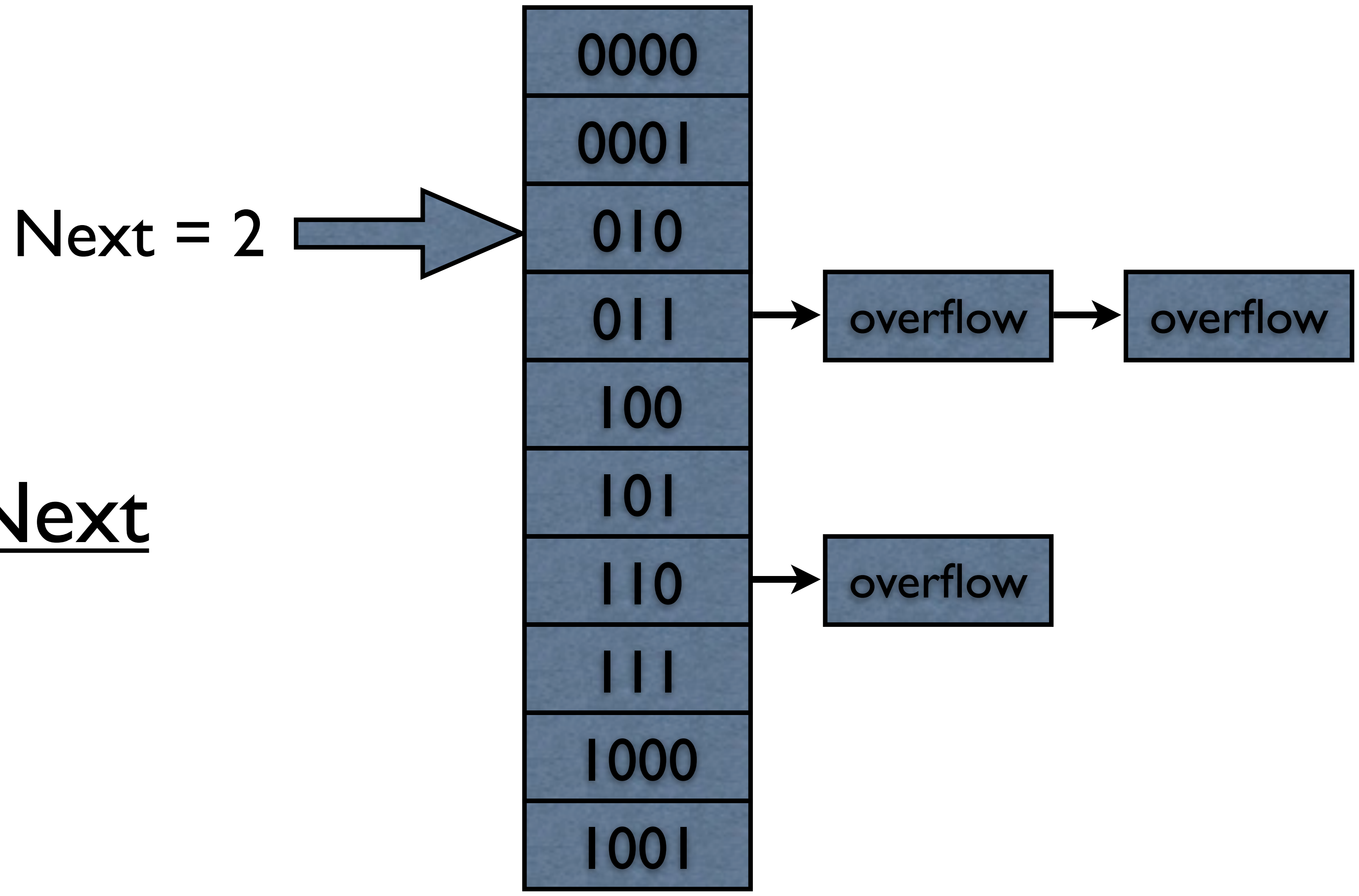
$h(k) \% 2^{(Level+1)}$

1 (1001) or 9 (1001)

Linear Hashing:Splitting

Level = 3 (2^3 = 8 Entries)

Split Next



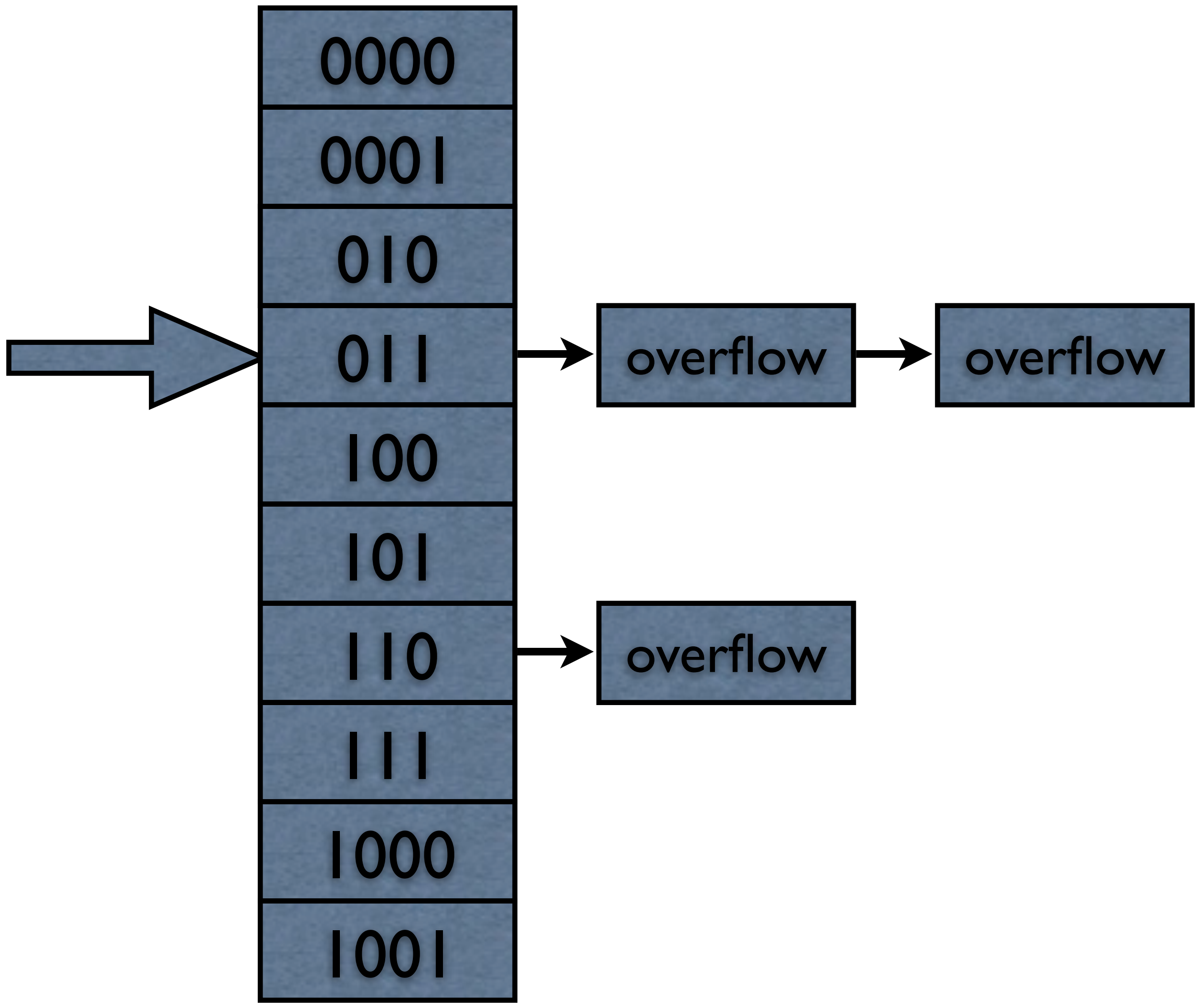
Linear Hashing:Splitting

Level = 3 (2^3 = 8 Entries)

Next = 2

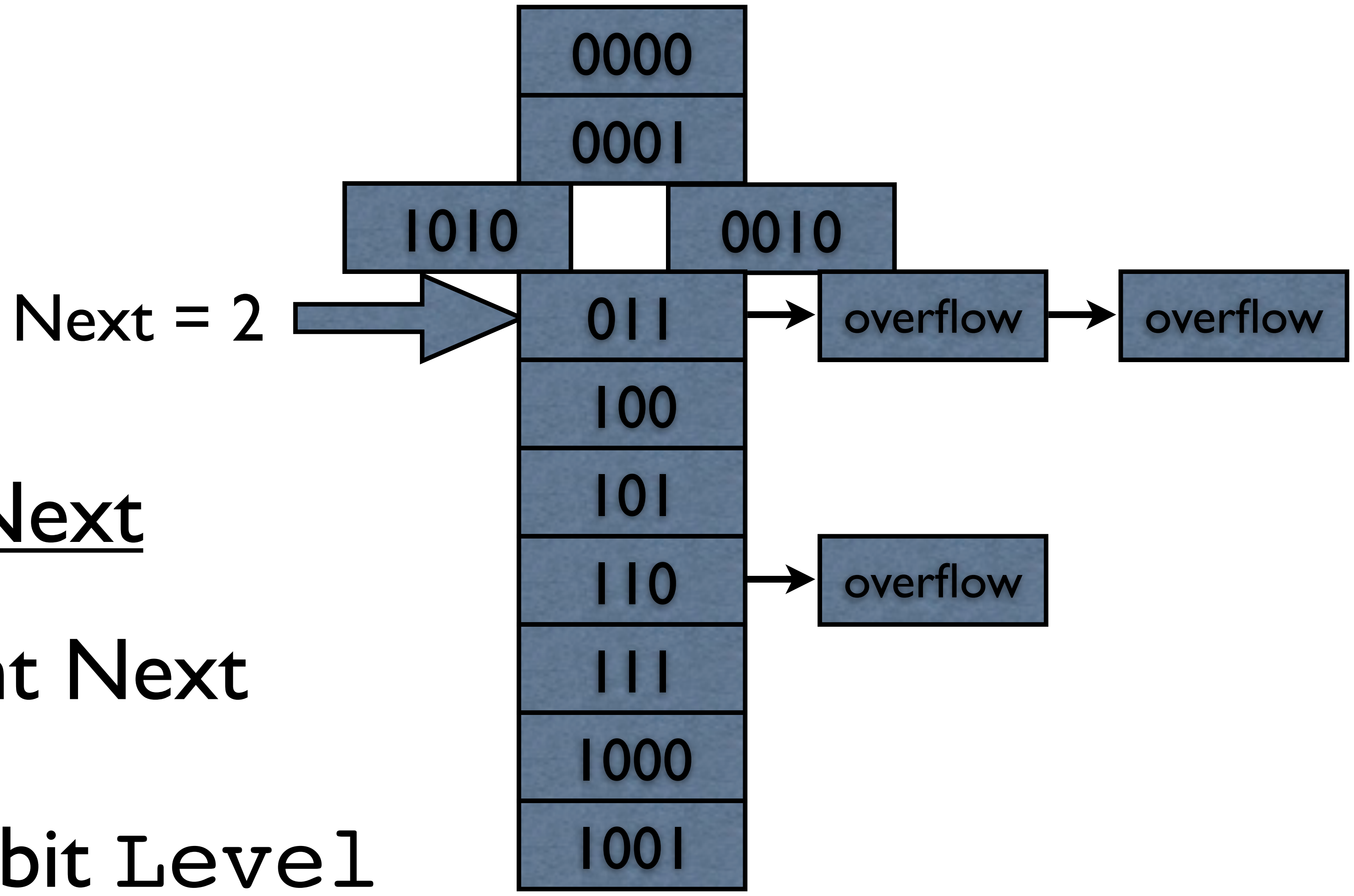
<u>Split Next</u>

Increment Next



Linear Hashing:Splitting

Level = 3 (2^3 = 8 Entries)



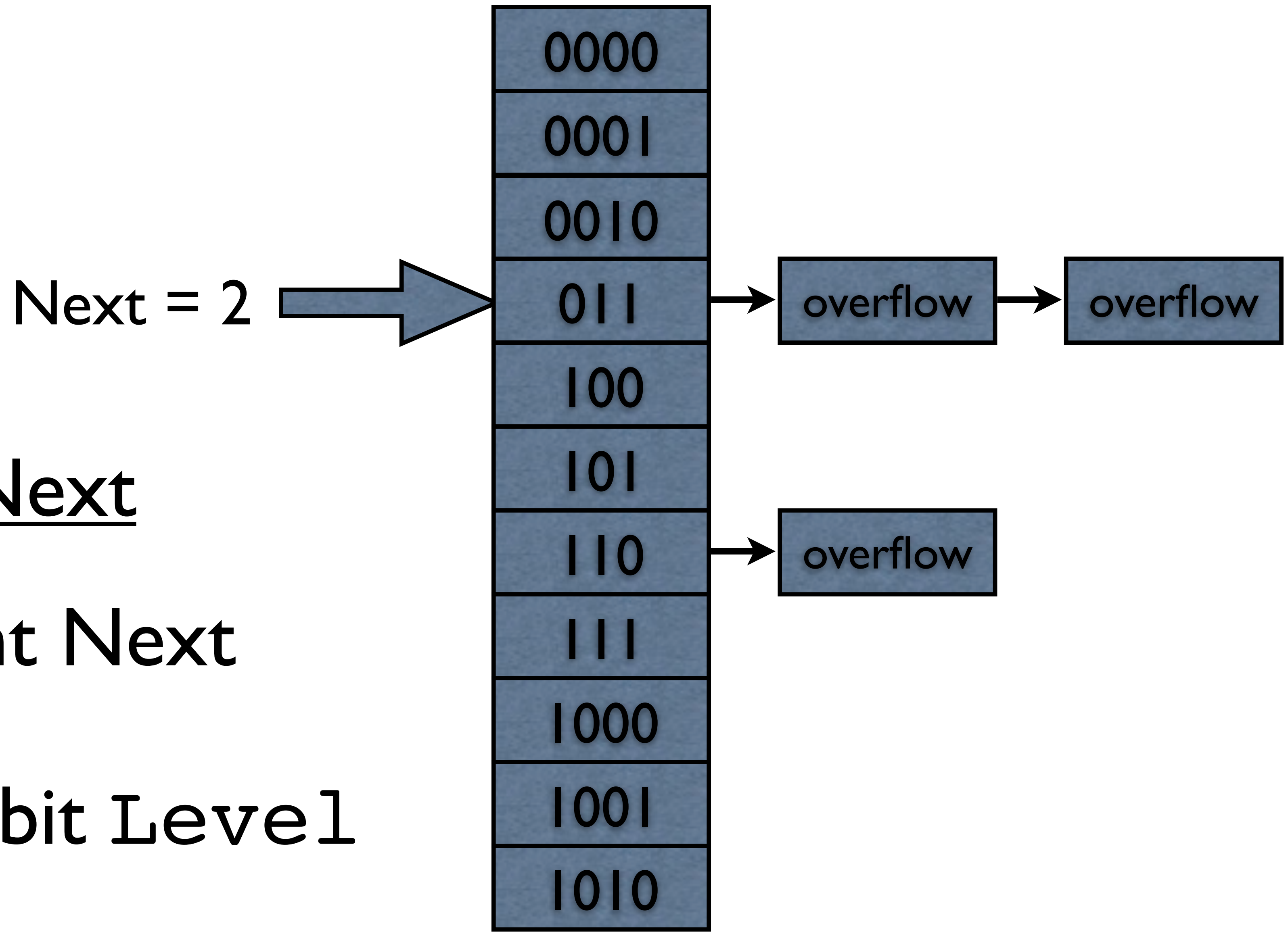
Split Next

Increment Next

Partition on bit `Level`

Linear Hashing:Splitting

Level = 3 (2^3 = 8 Entries)



<u>Split Next</u>

Increment Next

Partition on bit `Level`


Any Questions?

Linear Hashing

- When to we split?
    - It depends on the application.
- Whenever `Next` bucket is full
- After random insertions
- After a fixed number of insertions (size)
- Background process splits as needed.

Extendible vs Linear

- The two algorithms are actually quite similar.
  - Keep some data pages un-split
    - Minimize repartitioning required to split.
  - Use least-significant bits to ensure that new buckets will be appended to the end.
- Linear allocates buckets in sequential order.
  - Is this helpful? When/how?


Any Questions?

Time permitting...

Consistent Hashing

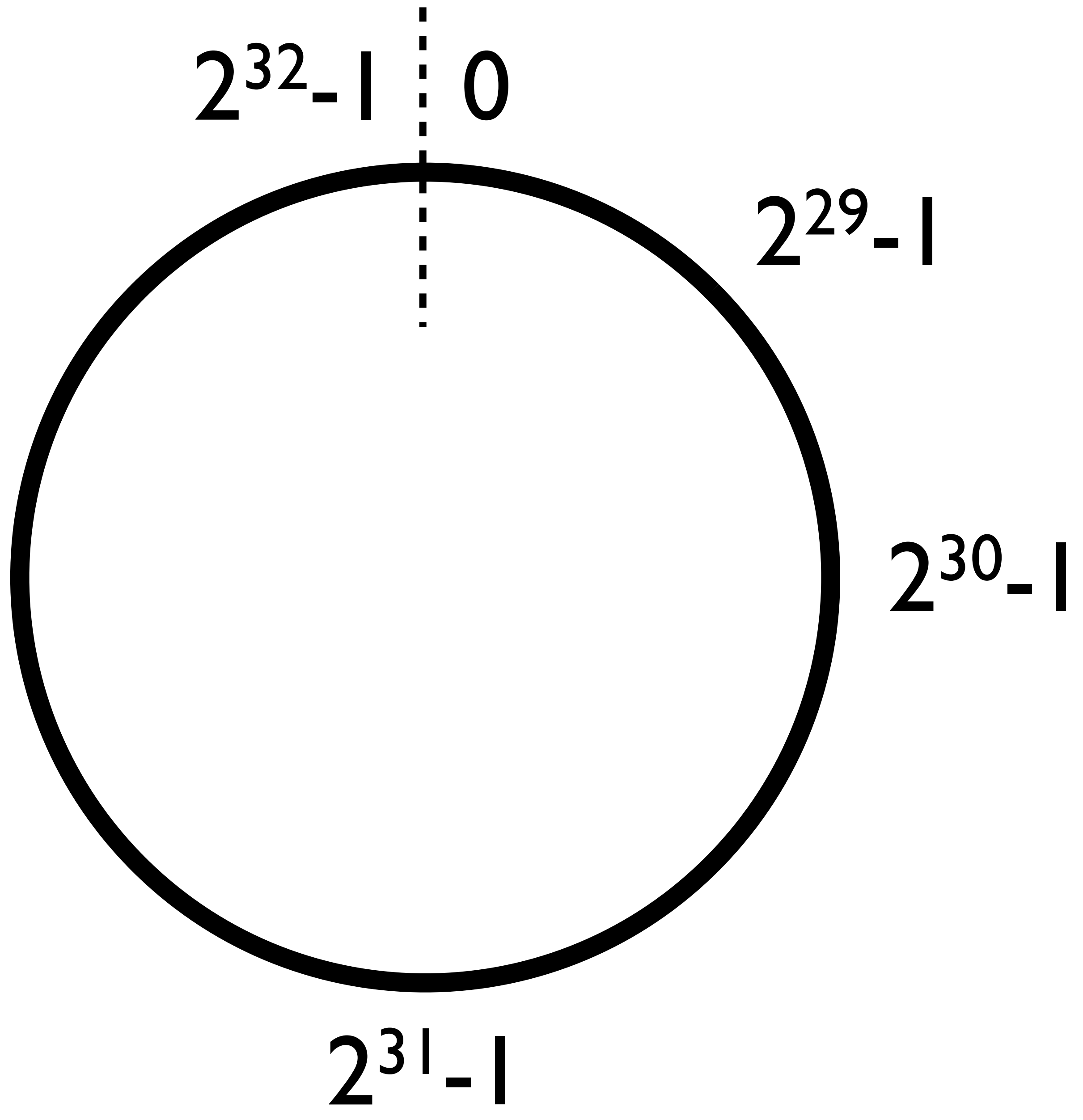
('Chord: A Scalable Peer-to-peer…', Stoica et al.)

- **Insight:** Make split/merge faster by making bin boundaries nondeterministic.
- Used mostly in distributed data-stores
  - (Amazon, Facebook, …)
  - Minimal applications to file-based storage.

Consistent Hashing

0 2^{32}-1

Consistent Hashing



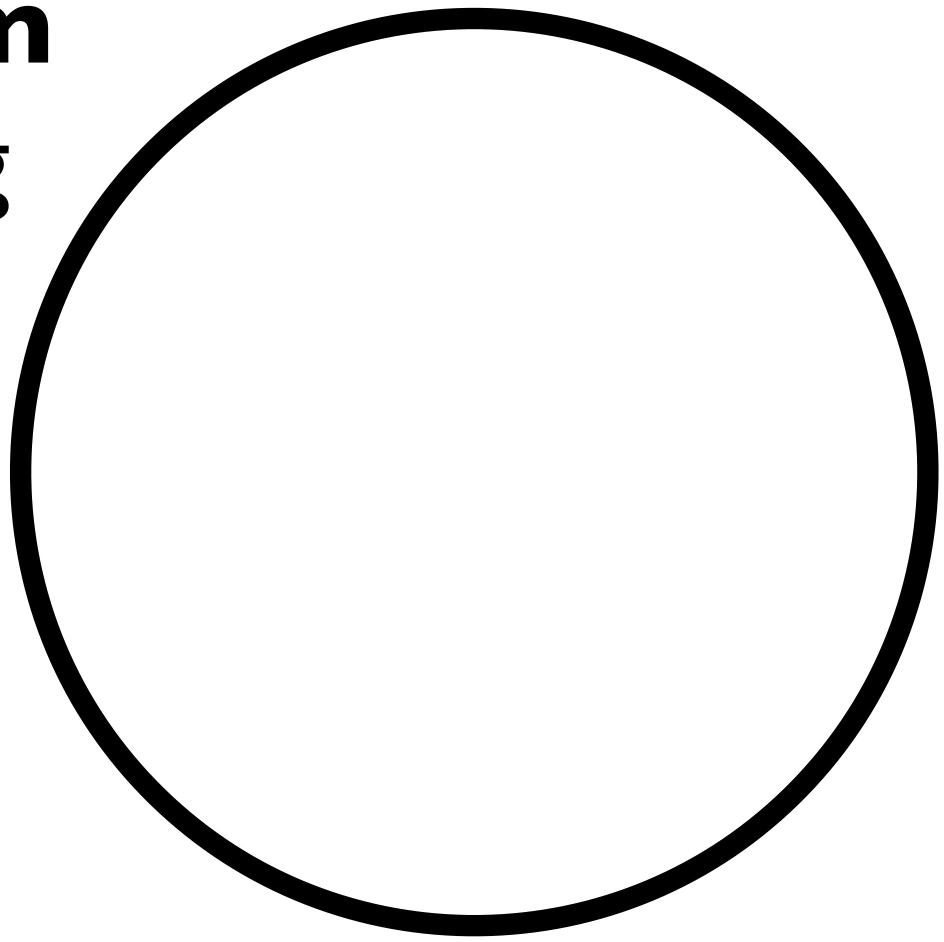
Modular Arithmetic (mod 2^{32})

$(2^{32}-1)+1 = 0$

Numbers form a 'Ring'

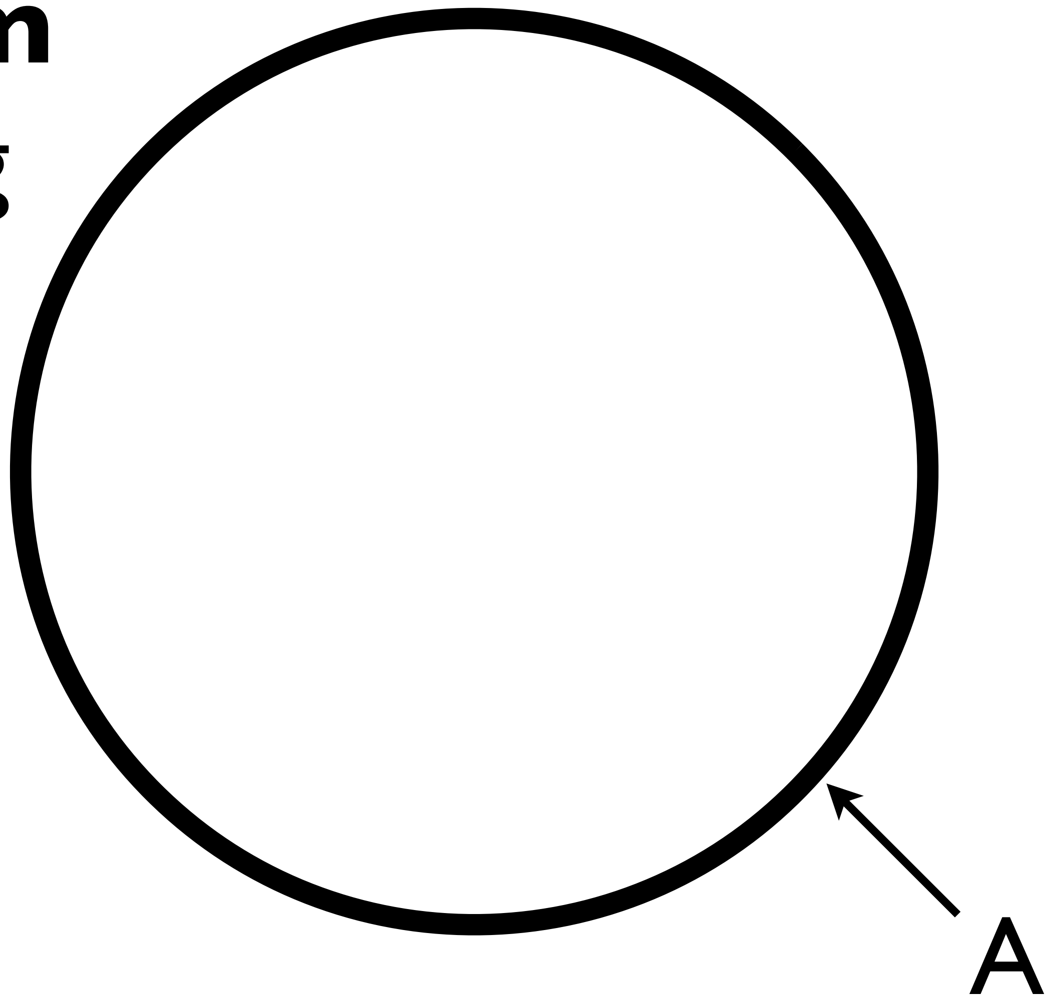
Consistent Hashing

Assign each
bucket a **random**
point on the ring



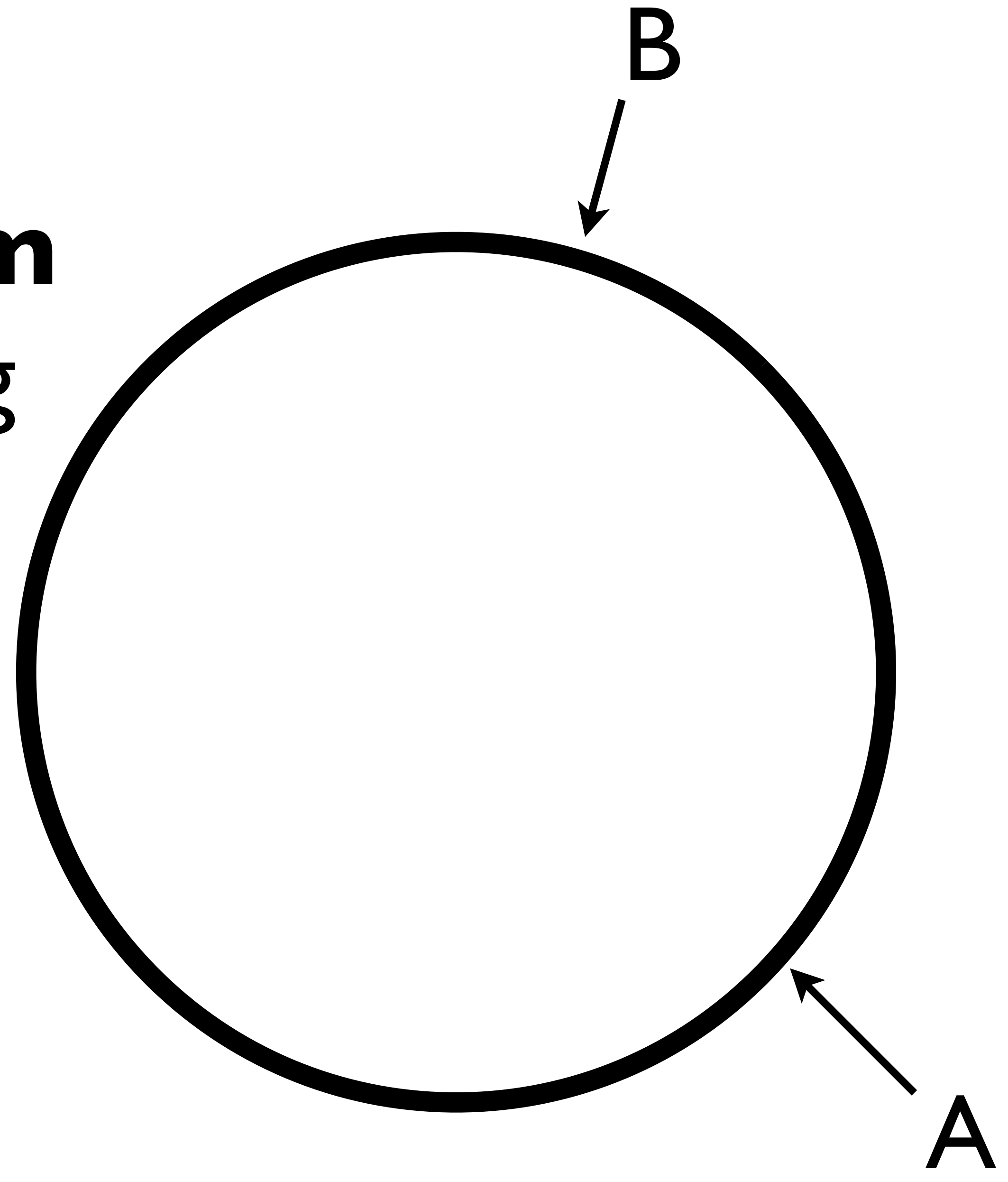
Consistent Hashing

Assign each bucket a **random** point on the ring



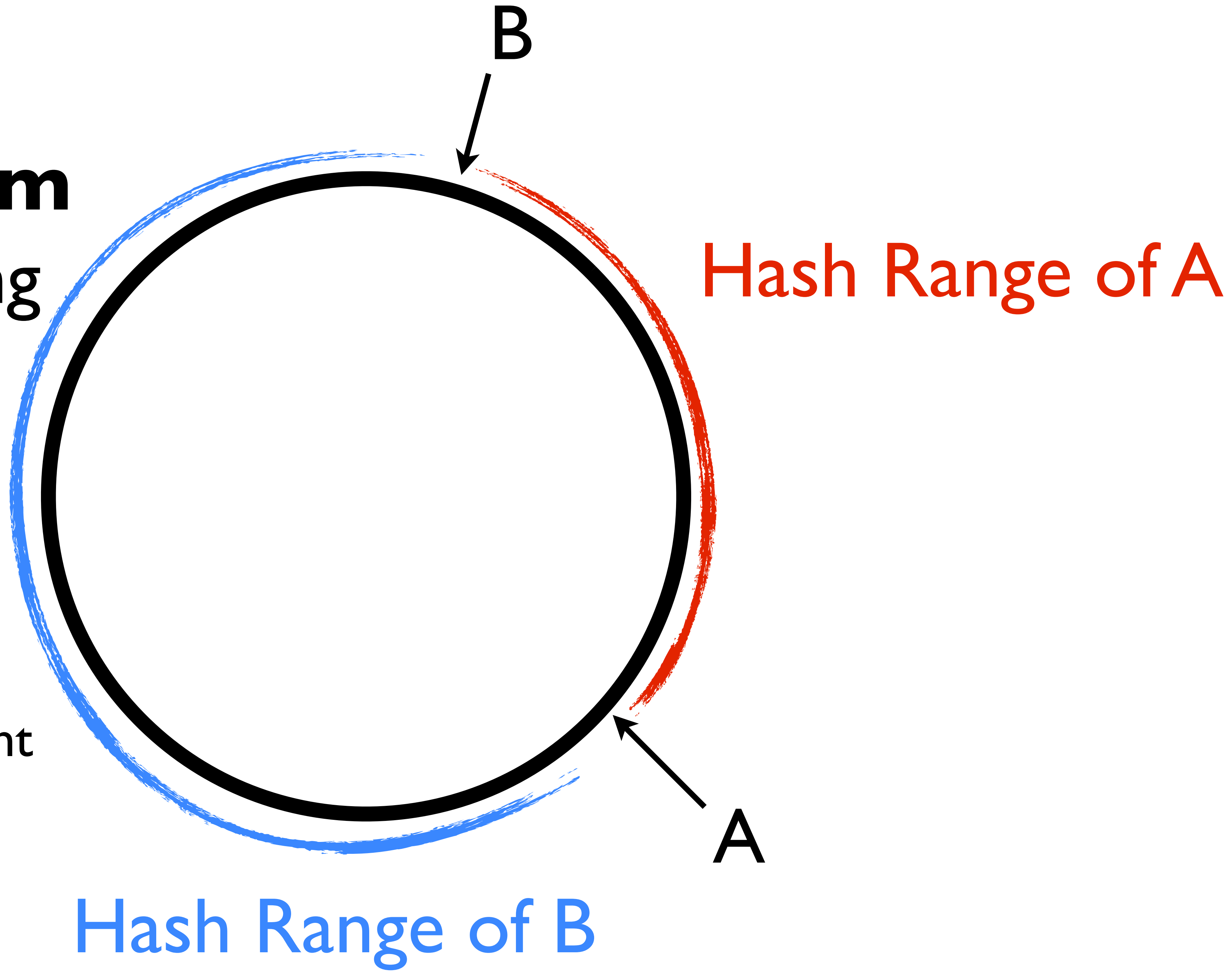
Consistent Hashing

Assign each bucket a **random** point on the ring



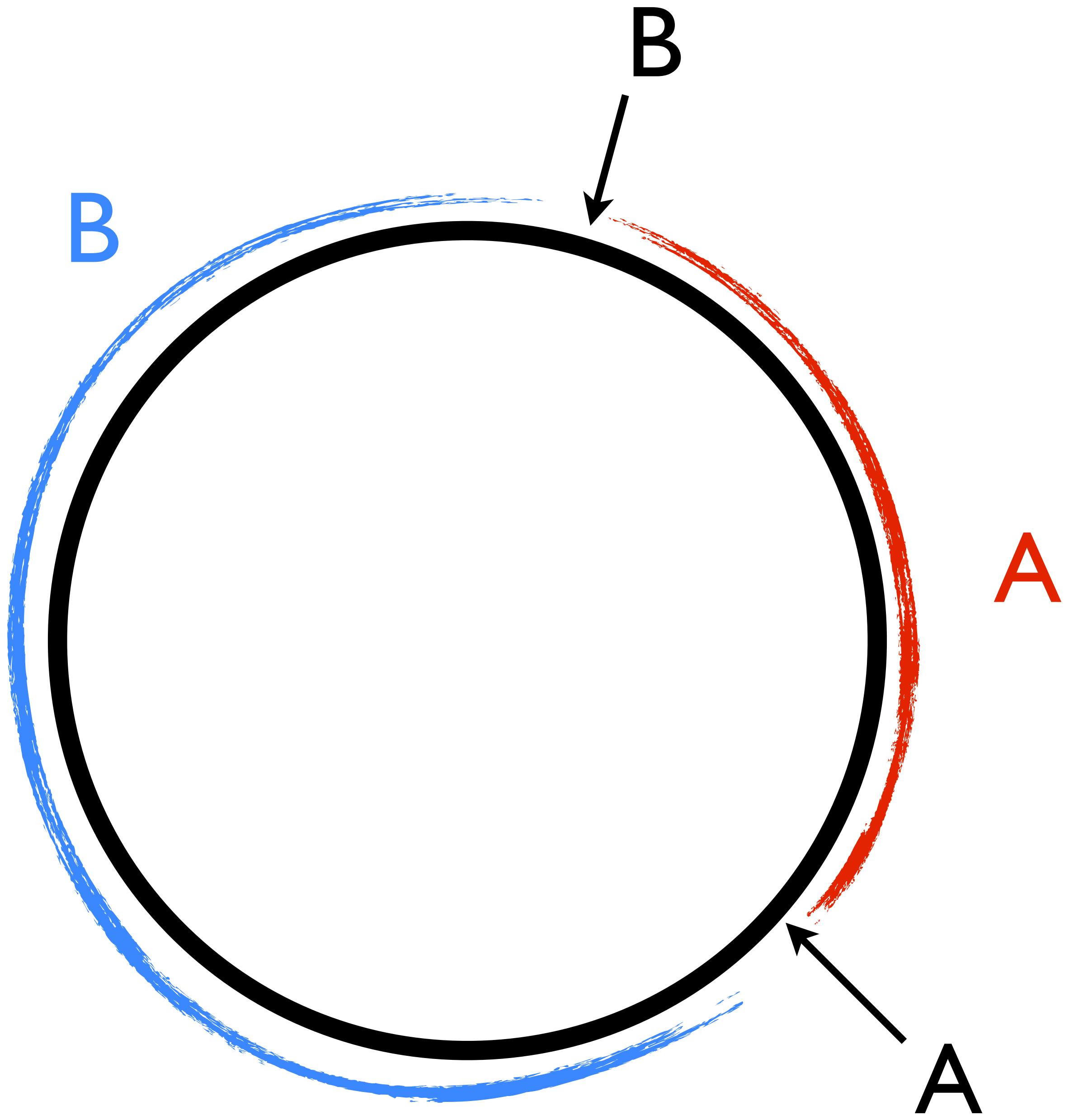
Consistent Hashing

Assign each bucket a **random** point on the ring

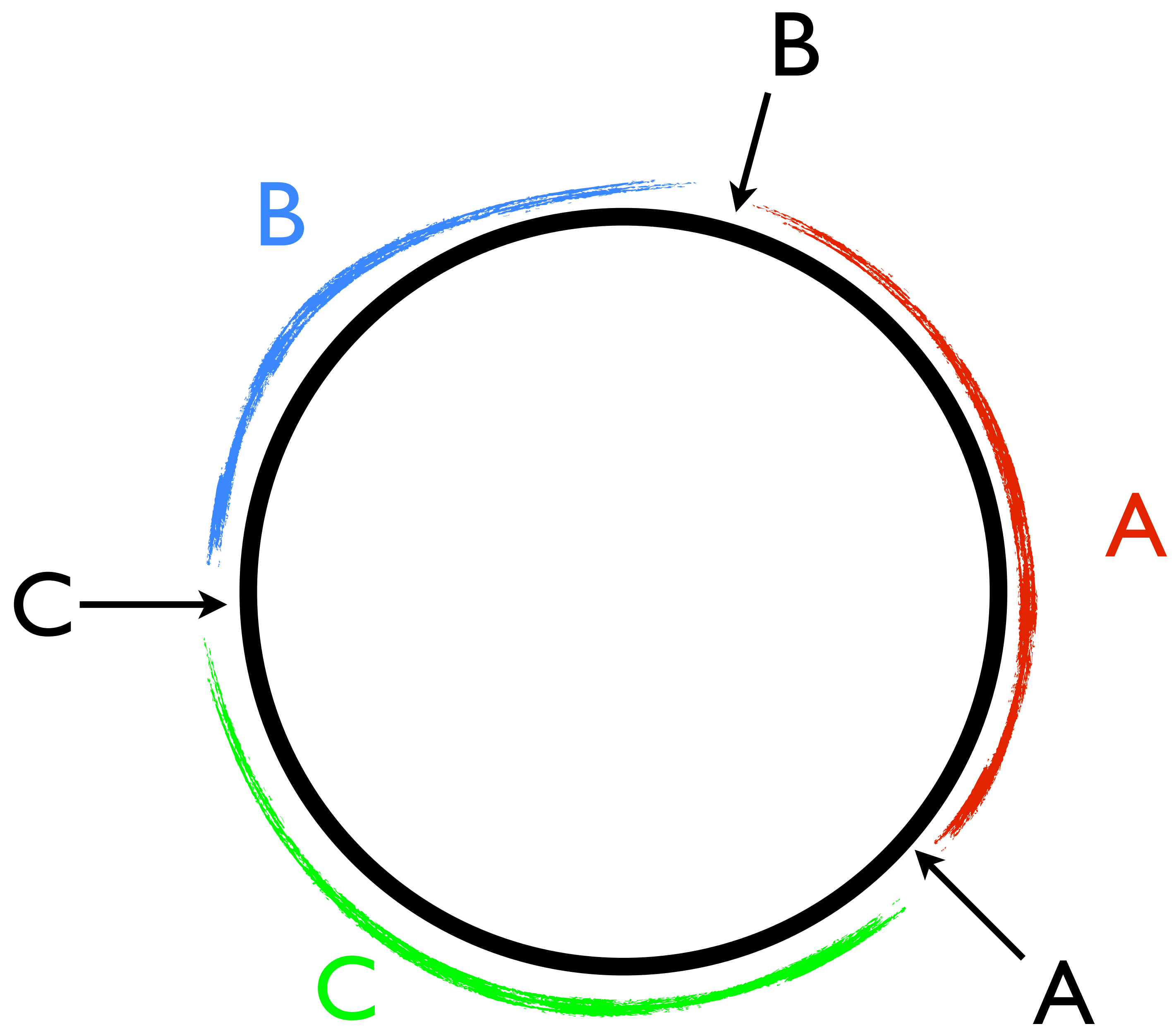


Each bucket contains values that hash to ring positions between its point and its **predecessor**

Consistent Hashing



Consistent Hashing



Consistent Hashing

- Splits/Merges are cheap.
  - At most 2 buckets are affected.
  - No need for page duplication.
- Mapping hash value to bucket is expensive.
  - Need to have a lookup mechanism/directory.
- **Chord**: Decentralized lookup mechanism.

If you're interested, the paper's a fun read
http://dl.acm.org/citation.cfm?id=383071&CFID=178000773&CFTOKEN=34551248


Any Questions?

Summary

- Size of a hash table is important
  - Too big: Wasted Space/IOs
  - Too small: Collisions/Overflow Pages
- Dynamic hashing requires carefully managing how data is repartitioned.