# RecSplit
# Minimal Perfect Hashing

A new algorithm that is faster and uses less space

# Contents

- Use Cases
- Hashing a Fixed Set
- Universal Hashing
- (Minimal) Perfect Hashing
- RecSplit
- BDZ, CHD

# Use Cases

- You have a large, fixed (immutable) set of keys
- Keys don't fit in memory (many millions)
- Values are small (e.g. 8 bits/entry, or possibly just one bit if you do garbage collection)
- You need a small in-memory index (e.g. 2 bits/key)

# Use Cases: Examples

- Page Rank: billions of URLs, each with an 8 bit counter, and need keep it in memory for speed
- Garbage collection in a de-duplication store: billion of keys, each with one bit, to mark used entries
- Eye-based tracking: each word has many thousand "near matches" (tracking is very inaccurate)

# Possible Solutions

- Hashing can have collisions, so we can't use it
- Perfect hashing: maps $n$ keys, without collision, to an index $0..m$, where $m>n$ (gaps are possible)
- Minimal perfect hashing: map $n$ keys, without conflicts and without gaps, to a index $0..n$
- $k$-perfect hashing: each key has at most $k$ collisions

# Hashing a Fixed Set

| | | Collisions | Gaps |
|---|---|---|---|
| | Regular Hashing | Yes | Yes |
| RecSplit | Perfect Hashing | No | Yes |
| RecSplit | Minimal Perfect Hashing | No | No |
| RecSplit | *k*-Perfect Hashing | at most *k* | No |

# Perfect Hashing

Disadvantages:

- Only works for static sets
- Requires O($n$) memory for the hash function description
- Keys are not stored; can not check if a key is in the set (if false positives are OK, can store a hash "fingerprint")

# Universal Hashing

- To generate a perfect hash function (PHF), can not rely on a simple hash function, due to possible conflicts
- Universal hashing can create endless many hash codes for each key, using an indexed (seeded) hash function
- All (minimal-) perfect hashing algorithms use universal hashing

# Universal Hashing

Regular hash function (Java):

```
public static int hashCode(byte a[]) {
    int result = 1;
    for (byte element : a)
        result = 31 * result + element;
    return result;
}
```

# Universal Hashing

Universal hash function (simplified):

```
public static int universalHash(byte a[], int index) {
    int result = index;
    for (byte element : a)
        result = 31 * result + element + index;
    return result;
}
```

- This sample implementation is not very "random"
- Better use more secure algorithms, such as seeded Murmur Hash, SipHash, SHA-256,...

# Universal Hashing

- Example: list of universalHash(key, index) mod 4

| Index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| **Key: "a"** | 1 | 2 | 3 | 2 | 3 | 2 | 3 | 2 |
| **Key: "b"** | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 1 |
| **Key: "c"** | 0 | 3 | 2 | 2 | 3 | 1 | 1 | 0 |
| **Key: "d"** | 2 | 1 | 2 | 2 | 0 | 1 | 0 | 0 |

# Universal Hashing

- f(k) = universalHash(k, 6) mod 4
  is a MPHF (minimal perfect hash function)

| Index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Key: "a" | 1 | 2 | 3 | 2 | 3 | 2 | 3 | 2 |
| Key: "b" | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 1 |
| Key: "c" | 0 | 3 | 2 | 2 | 3 | 1 | 1 | 0 |
| Key: "d" | 2 | 1 | 2 | 2 | 0 | 1 | 0 | 0 |

- Brute force algorithm: find the first index where
  universalHash(key, index) mod 4
  has no collision

# (Minimal) Perfect Hashing

- So generating an MPHF is simple for small sets: try universalHash(x), and if there is a collision, try with x+1,... until there is no collision

- But for larger sets, too many tries are needed (millions for sets of size > 20)

- The challenge is to support huge sets, O(n) generation, and constant time evaluation

# Best Algorithms

- Theoretical lower bound: 1.44 bits/key
- RecSplit: around 1.8 bits/key
- CHD: around 2.1 bits/key
- BDZ: around 2.7 bits/key

(assuming somewhat reasonable generation time; both RecSplit and CHD can approach the theoretical lower bound given enough time)

# RecSplit

The algorithm has three phases:

- Partitioning
- Bucket Processing
- Storing

# RecSplit Partitioning

A. Partition the *n* keys into *n*/100 buckets

B. Overflow: buckets with more than 2000 keys are merged and processed with BDZ (fallback). This is an  extremely rare case, and makes RecSplit a *pseudo* hybrid algorithm: less than 0.0000...001% of the keys fall into this category (in practise: none).

C. What remains is bucket with up to 2000 keys

# RecSplit Partitioning

| Keys | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | jan | feb | mar | apr | may | jun | jul | aug | sep | oct | nov | dec |

# RecSplit Partitioning

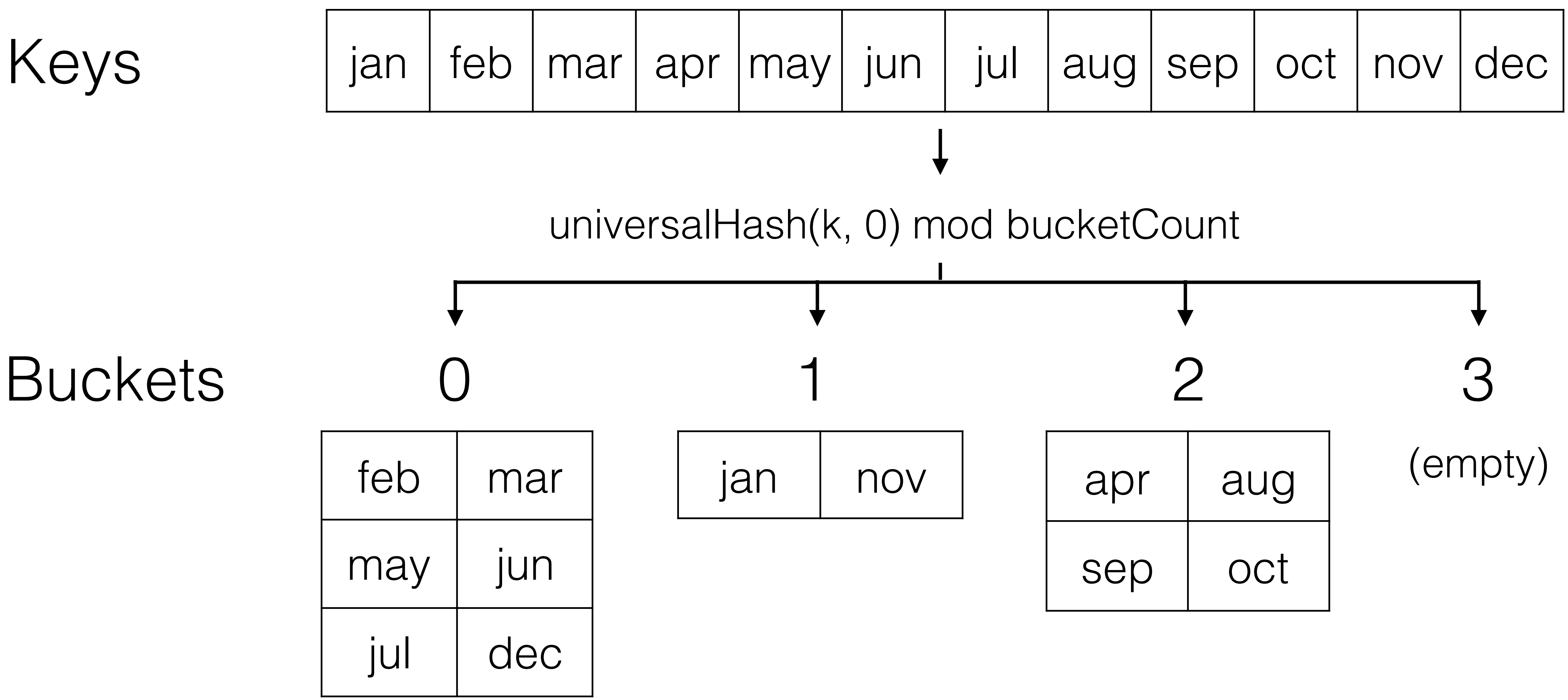
Keys

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| jan | feb | mar | apr | may | jun | jul | aug | sep | oct | nov | dec |

universalHash(k, 0) mod bucketCount

Buckets 0 1 2 3

# RecSplit Partitioning



# RecSplit Partitioning

Keys

| jan | feb | mar | apr | may | jun | jul | aug | sep | oct | nov | dec |
|---|---|---|---|---|---|---|---|---|---|---|---|

universalHash(k, 0) mod bucketCount

Buckets

0

| feb | mar |
|---|---|
| may | jun |
| jul | dec |

1

| jan | nov |
|---|---|

2

| apr | aug |
|---|---|
| sep | oct |

3

(empty)

Overflow (empty, as no bucket is overly large)

# RecSplit Bucket

Convert each bucket into a tree recursively:

A. Sets of size 0 and 1: no processing is needed

B. Sets up to size 8 (or so): use the "brute force" algorithm described in Universal Hashing

C. Larger (size *s*): find the first x so that universalHash(key, x) splits the sets in two (sizes *s*/2 and rest), and process each set recursively

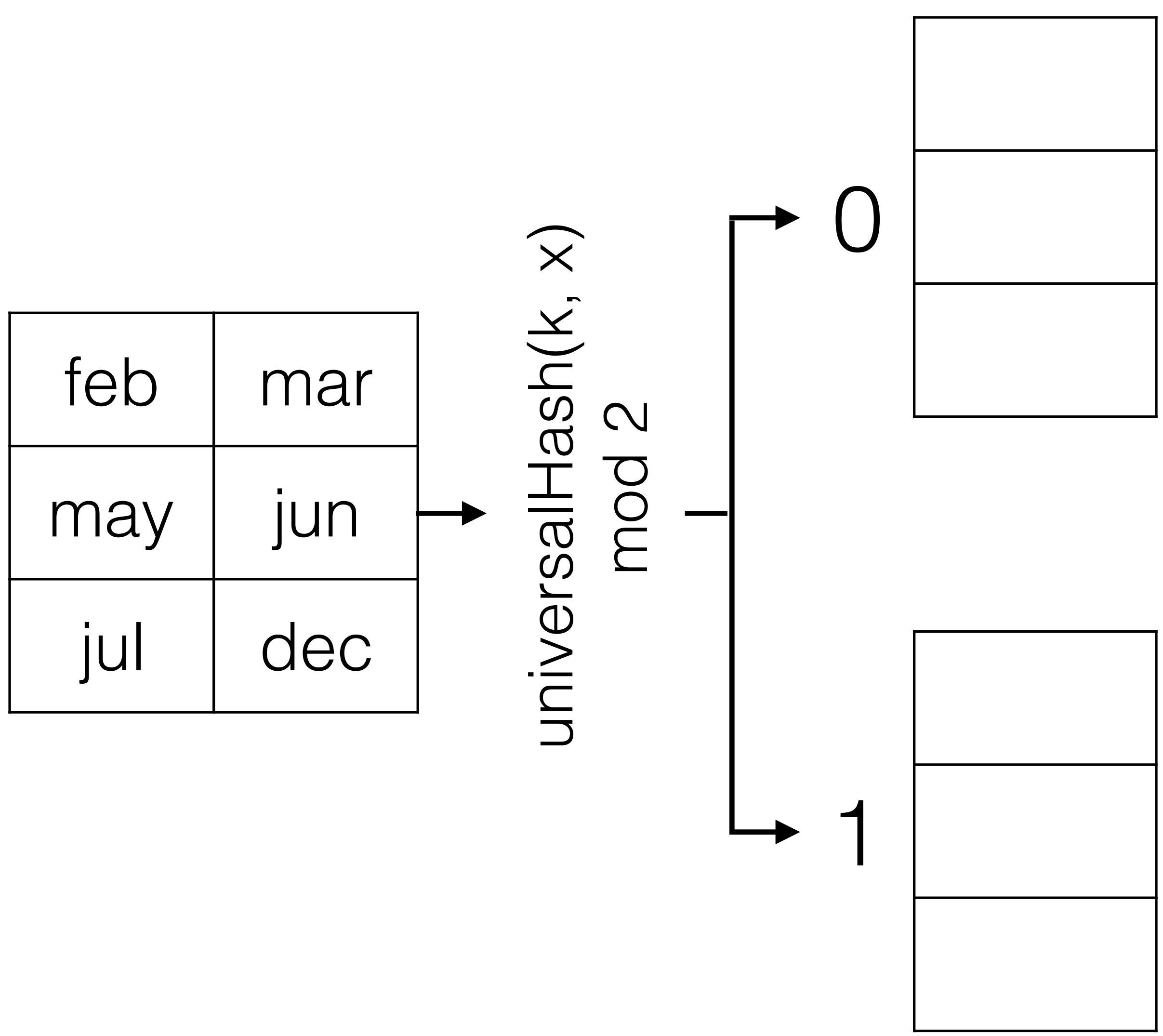
# RecSplit Bucket

Bucket

| | |
|---|---|
| feb | mar |
| may | jun |
| jul | dec |

# RecSplit Bucket

## Bucket

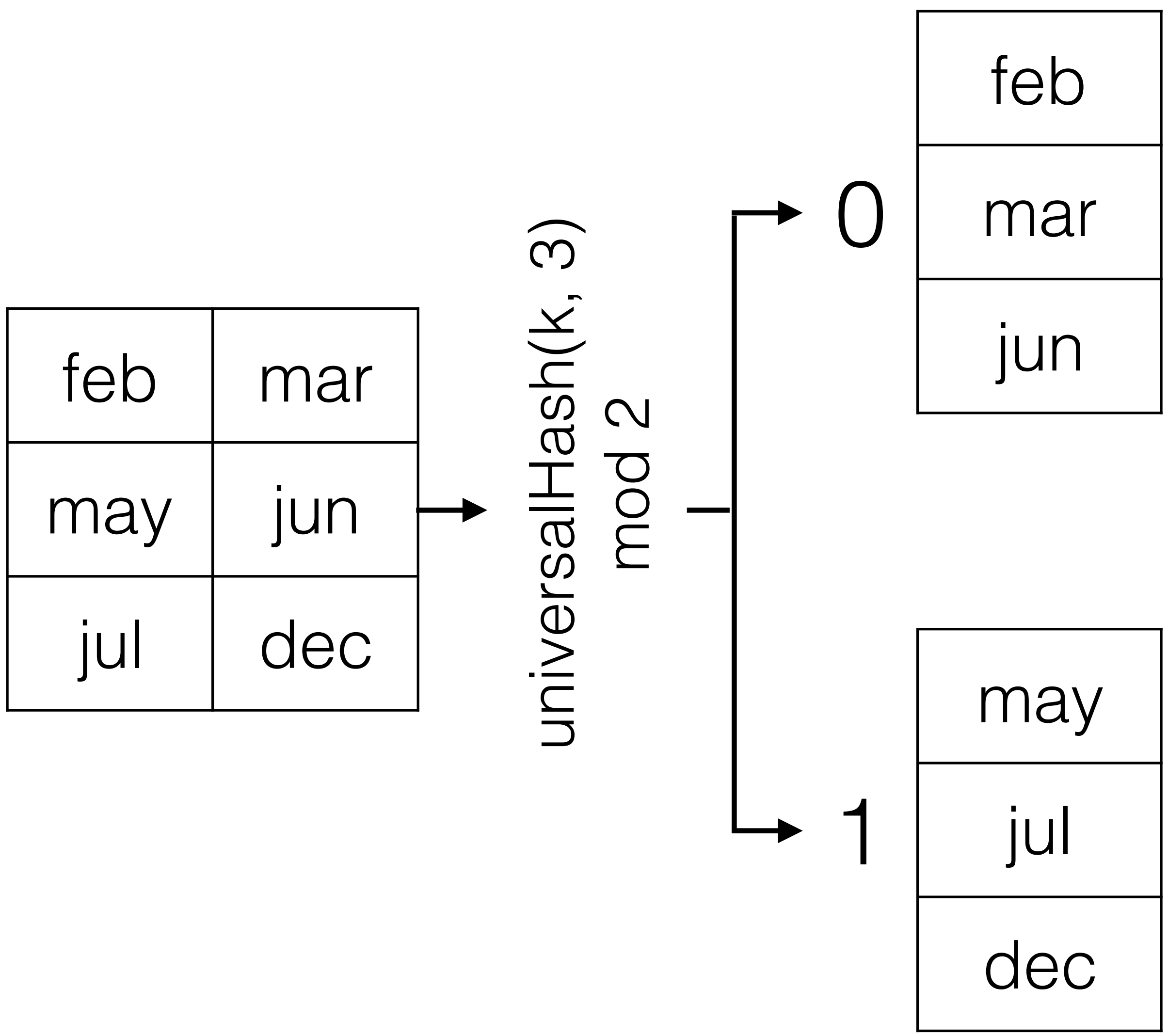



Search the first
x that evenly splits
the set

(for an odd size,
the first subset has
one entry more)

# RecSplit Bucket

Bucket    Bucket Description: 3,

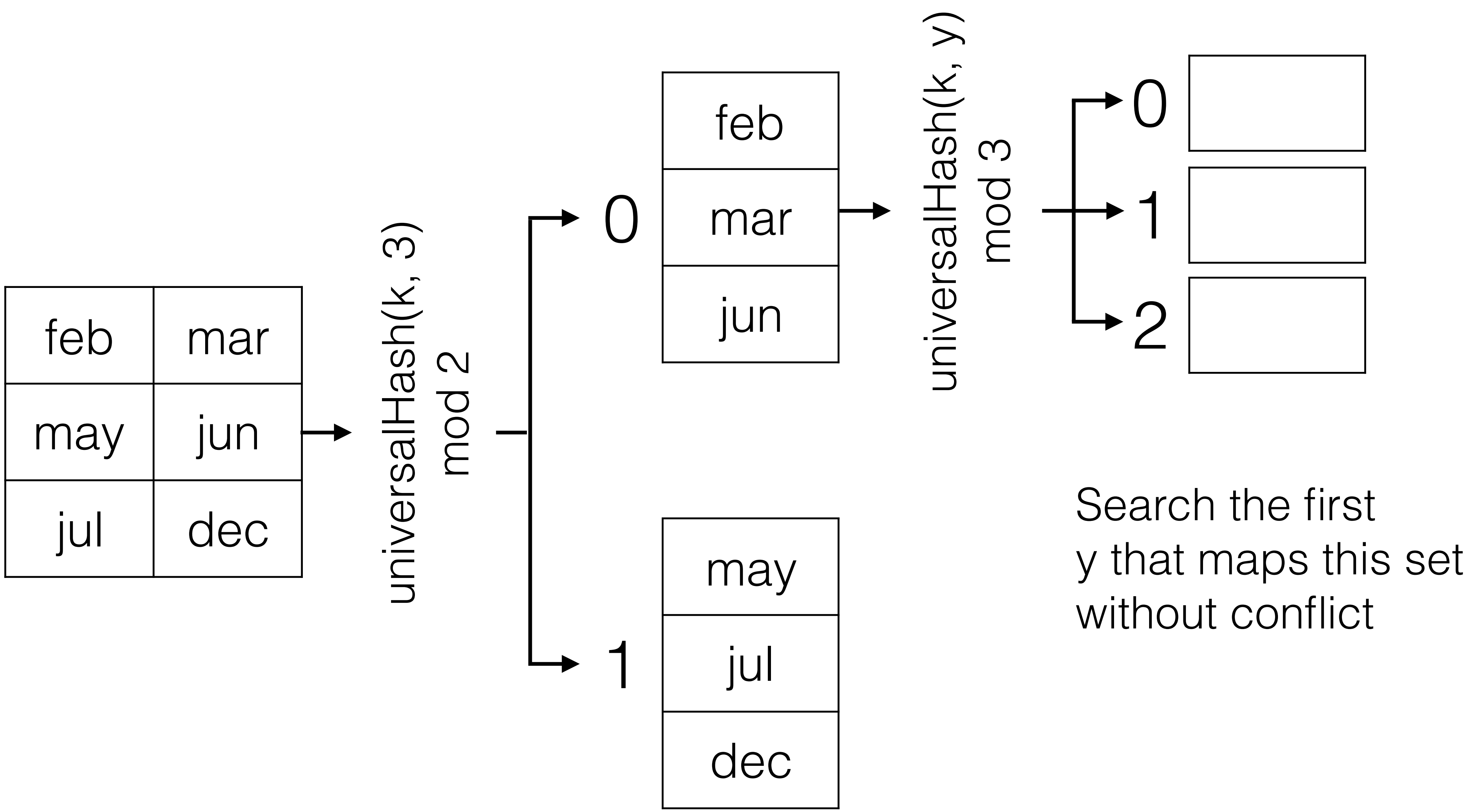



We find that
universalHash(k, 3)
splits the set as needed,
so 3 is the first entry
in the bucket description

# RecSplit Bucket

Bucket

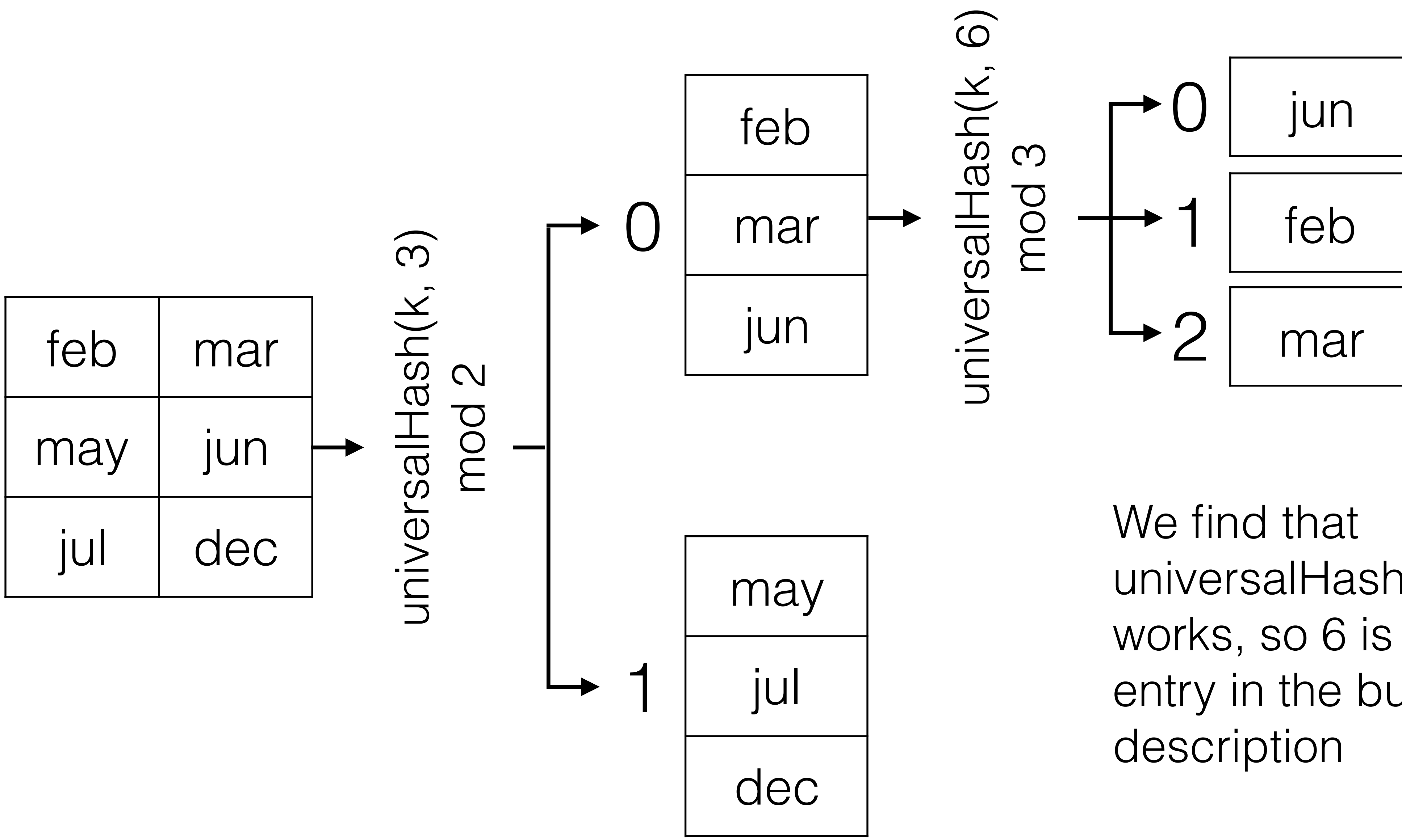
Bucket Description: 3,



# RecSplit Bucket

Bucket

Bucket Description: 3, 6,

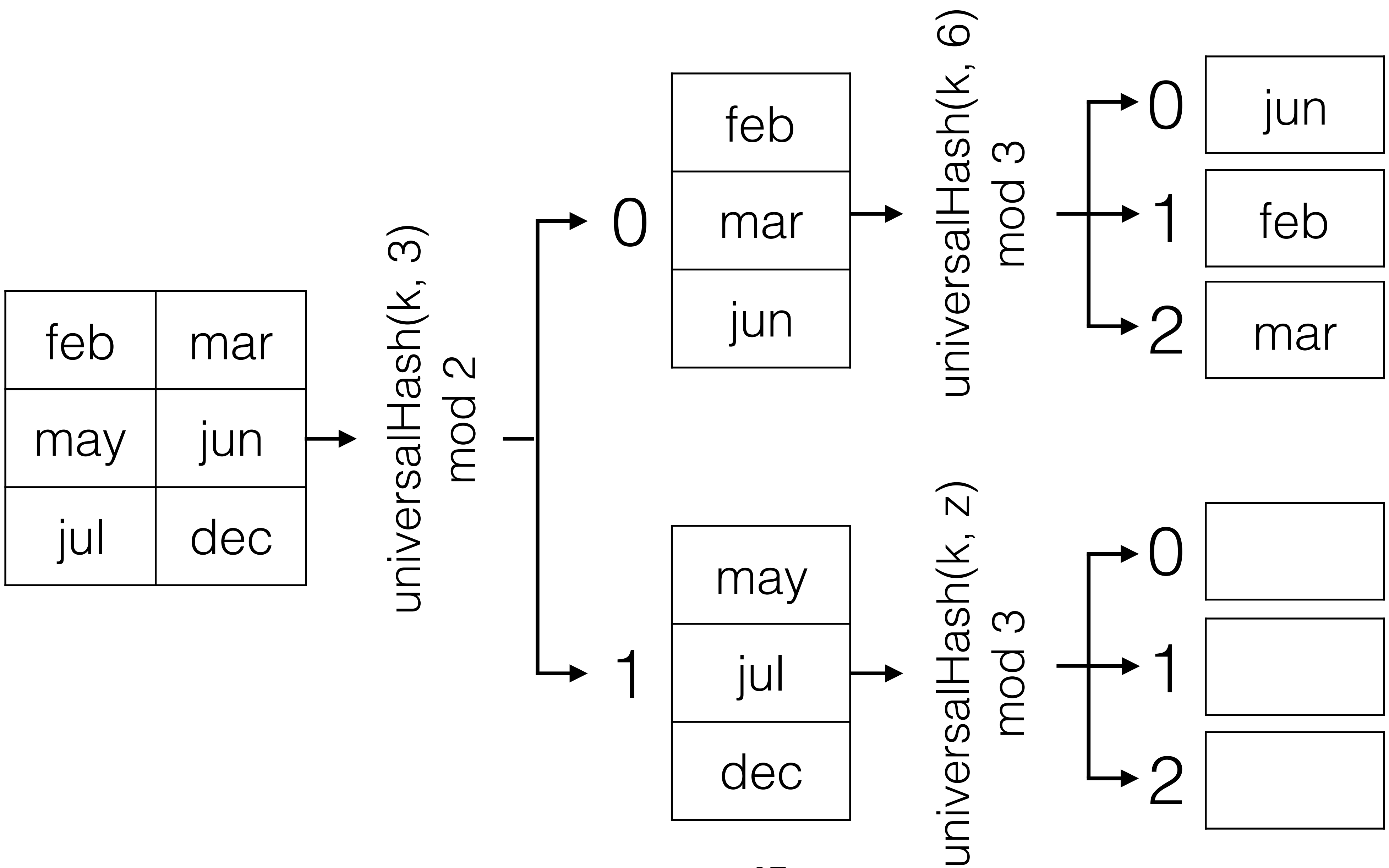



We find that universalHash(k, 6) works, so 6 is the next entry in the bucket description

# RecSplit Bucket

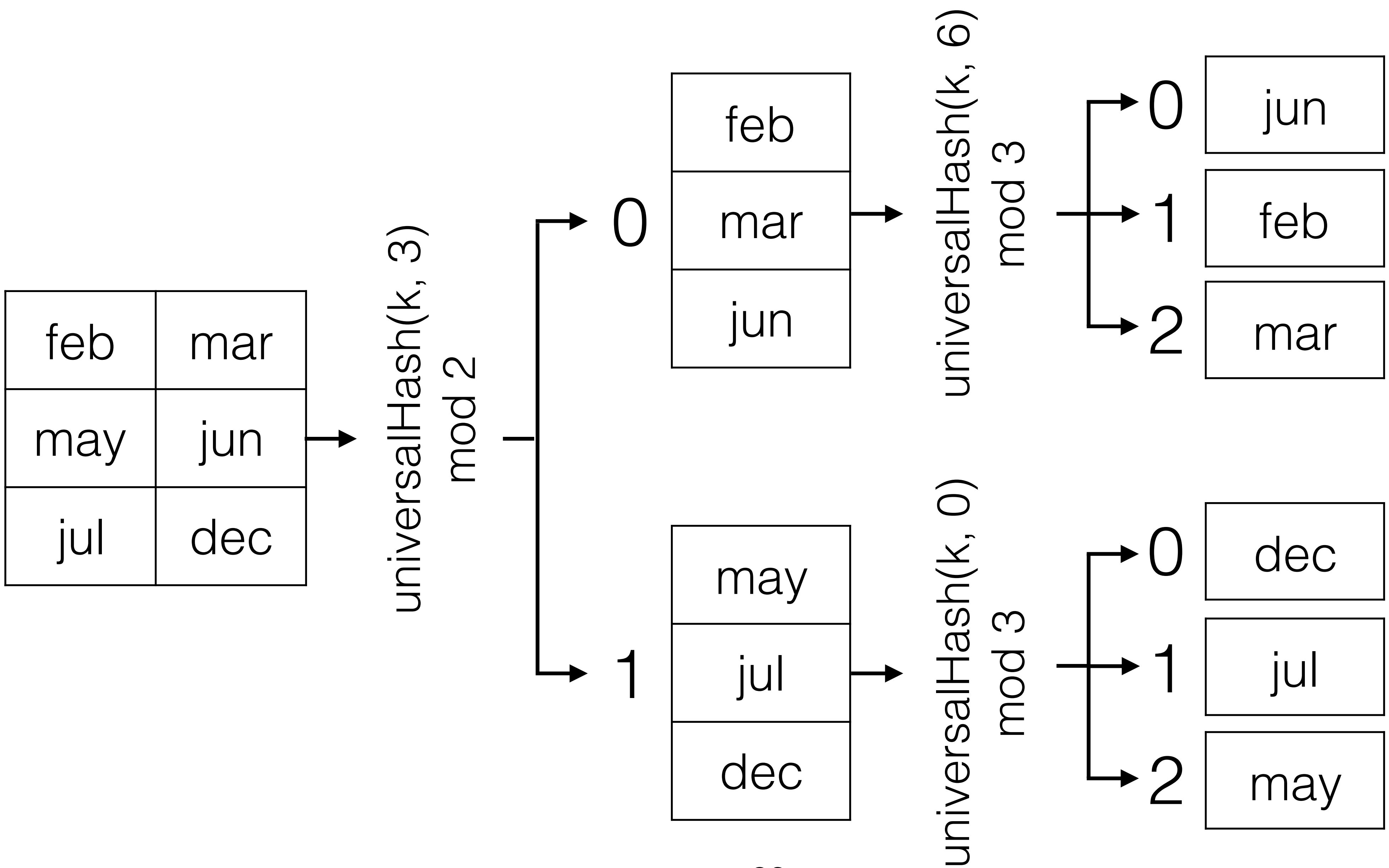
Bucket

Bucket Description: 3, 6,

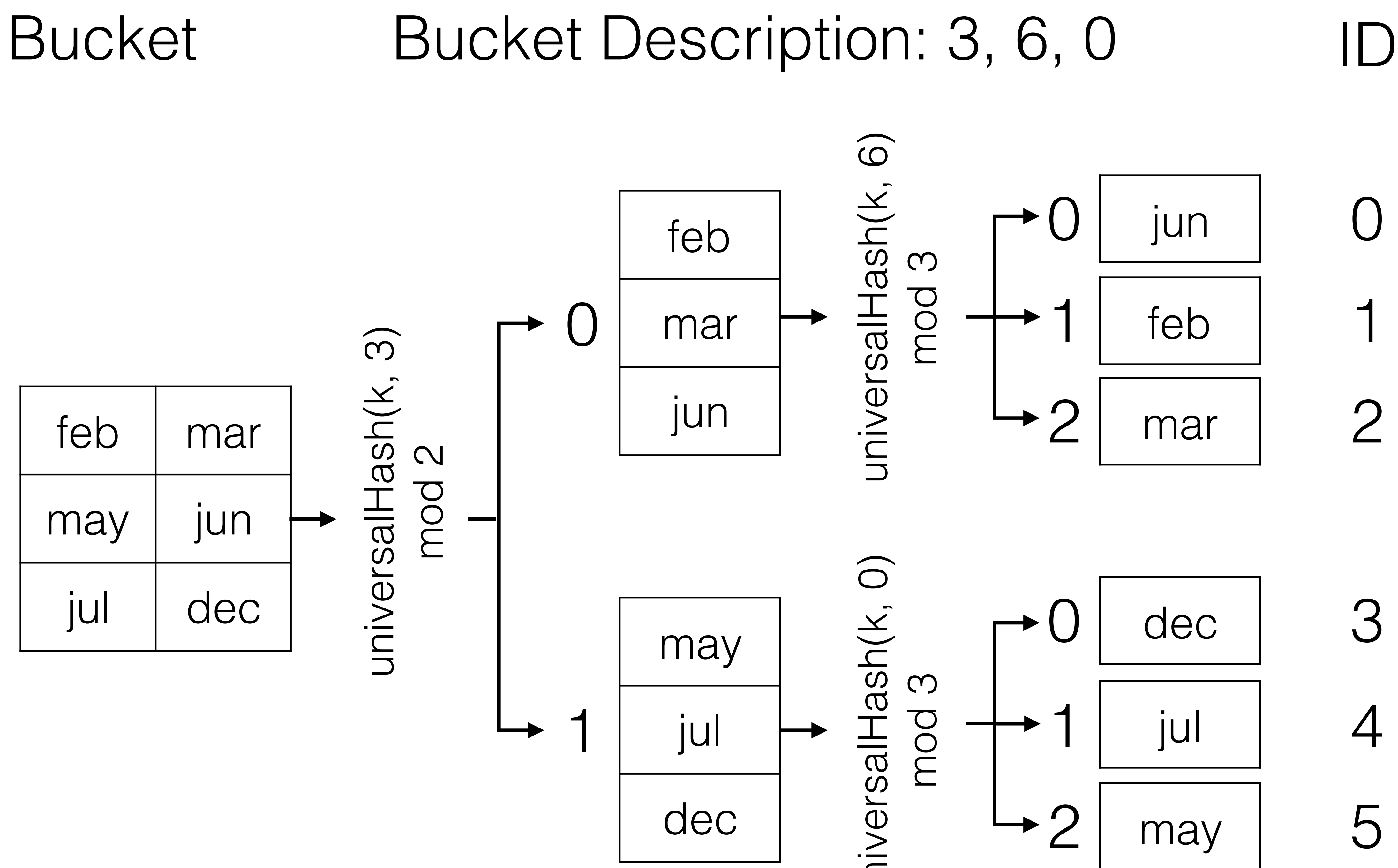


# RecSplit Bucket

Bucket    Bucket Description: 3, 6, 0



# RecSplit Bucket



# RecSplit Storing

A. Store the set size (using the Elias Delta code)

B. Store the bucket lookup table and bucket size table (as Elias-Fano monotone lists)

C. Store each bucket description: A bucket description is the list of universal hash indexes that describe how to split the tree
(stored using the Rice code)

# RecSplit Observations

- RecSplit is easy to parallelise as buckets are processed independently
- The average bucket size, and how to process buckets, is configurable, which allows to trade space for generation time and evaluation speed

# RecSplit (*k*-)Perfect

RecSplit is a MPHF (minimal perfect hash function), but small modifications allow for:

- (non-minimal) perfect hash, by changing the last stage to use a larger modulo
- *k*-perfect hash, by changing the last stage to use a smaller modulo and allow for *k* entries per slot

# BDZ http://cmph.sourceforge.net/bdz.html

A. For *n* keys, prepare 1.23*n* slots

B. For each key, calculate 3 potential slots $h_0$, $h_1$, $h_2$

C. For each slot that contains only one key,
remove the key from the *other* slot that also contain it,
leaving just one key per slot

D. This will probably work; if not, repeat at B with
a new universalHash index

E. For each slot, store whether it contains $h_0$ $h_1$, or $h_2$

F. Use a Rank data structure to map the 1.23*n* slots to 0..*n*

# BDZ Observations

- Related to Cockoo Hashing
- Hard to parallelise because slots need to be processed in order
- If it doesn't work ($p<0.5$), start from scratch
- First generate a perfect hash function (PHF), and then map that to a minimum perfect hash function (MPHF) using a rank data structure
- Needs much more space than RecSplit

# CHD http://cmph.sourceforge.net/chd.html

A. Partition the *n* keys into *n*/4 buckets

B. Prepare 1.01*n* slots

C. For each bucket, starting with the largest,
   try mapping its keys to empty slots using
   universalHash(k, x) mod slotCount

D. If any slot is occupied, repeat at C

E. For each bucket, store the universalHash x used

F. Use a Rank data structure to map the 1.01*n* slots to 0..*n*

# CHD Observations

- Hard to parallelise because buckets are not independent, and need to be processed in order
- Probabilistic (try-until-it-works), but usually no need to start from scratch as with BDZ
- Generate a PHF, then map to MPHF (as BDZ)
- Still needs more space than RecSplit