Linear Hashing

Linear Hashing

- Central idea of hashing:
  - Calculate the location of the record from the key
  - Hash functions:
    - Can be made indistinguishable from random function
      - SH3, MD5, …
      - Often simpler
        - ID modulo slots

Linear Hashing

- Can lead to collisions:
  - Two different keys map into the same address
  - Two ways to resolve:
    - **Open Addressing**
      - Have a rule for a secondary address, etc.
    - **Chaining**
      - Can store more than one datum at an address

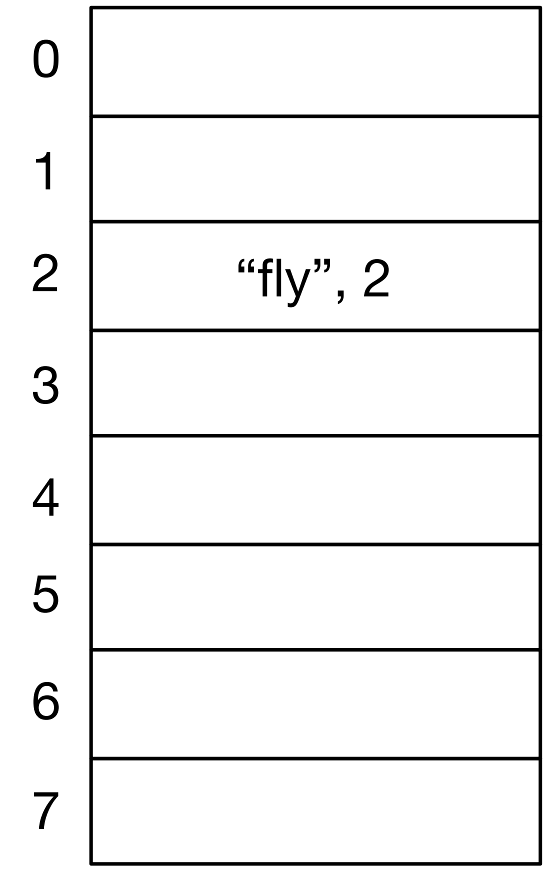
Linear Hashing

- Open addressing example:
  - Linear probing: Try the next slot

Hashing Example

```
def hash(a_string):
    accu = 0
    i = 1
    for letter in a_string:
        accu += ord(letter)*i
        i+=1
    return accu % 8
```

Insert “fly”



Hashing Example

```
def hash(a_string):
    accu = 0
    i = 1
    for letter in a_string:
        accu += ord(letter)*i
        i+=1
    return accu % 8
```

0	
1	
2	“fly”, 2
3	“gnu”, 2
4	
5	
6	
7	

Insert “gnu”

`hash(“gnu”) —> 2`

`Since spot 2 is taken, move to the`

Hashing Example

```
def hash(a_string):
    accu = 0
    i = 1
    for letter in a_string:
        accu += ord(letter)*i
        i+=1
    return accu % 8
```

0	
1	
2	“fly”, 2
3	“gnu”, 2
4	“hog”, 3
5	
6	
7	

Insert “hog”

`hash(“hog”) –> 3`

`Since spot is taken, move to the nex`

Hashing Example


```
def hash(a_string):
    accu = 0
    i = 1
    for letter in a_string:
        accu += ord(letter)*i
        i+=1
    return accu % 8
```

0	
1	
2	“fly”, 2
3	“gnu”, 2
4	“hog”, 3
5	
6	
7	“pig”, 7

Looking for “gnu”

`hash(“gnu”) –> 2`

`Try out location 2.  Occupied, but`

Hashing Example


```
def hash(a_string):
    accu = 0
    i = 1
    for letter in a_string:
        accu += ord(letter)*i
        i+=1
    return accu % 8
```

0	
1	
2	“fly”, 2
3	“gnu”, 2
4	“hog”, 3
5	
6	
7	“pig”, 7

Looking for “gnu”

`hash(“gnu”) —> 2`

`Try out location 3.  Find “gnu”`

Hashing Example


```
def hash(a_string):
    accu = 0
    i = 1
    for letter in a_string:
        accu += ord(letter)*i
        i+=1
    return accu % 8
```

Looking for “ram”

```
hash(“ram”) -> 3
```

0	
1	
2	“fly”, 2
3	“gnu”, 2
4	“hog”, 3
5	
6	
7	“pig”, 7

```
Look at location 3:  someone else is there
Look at location 4: someone else is there
Look at location 5: nobody is there, so if it wer
                                   dictionary, it w
```

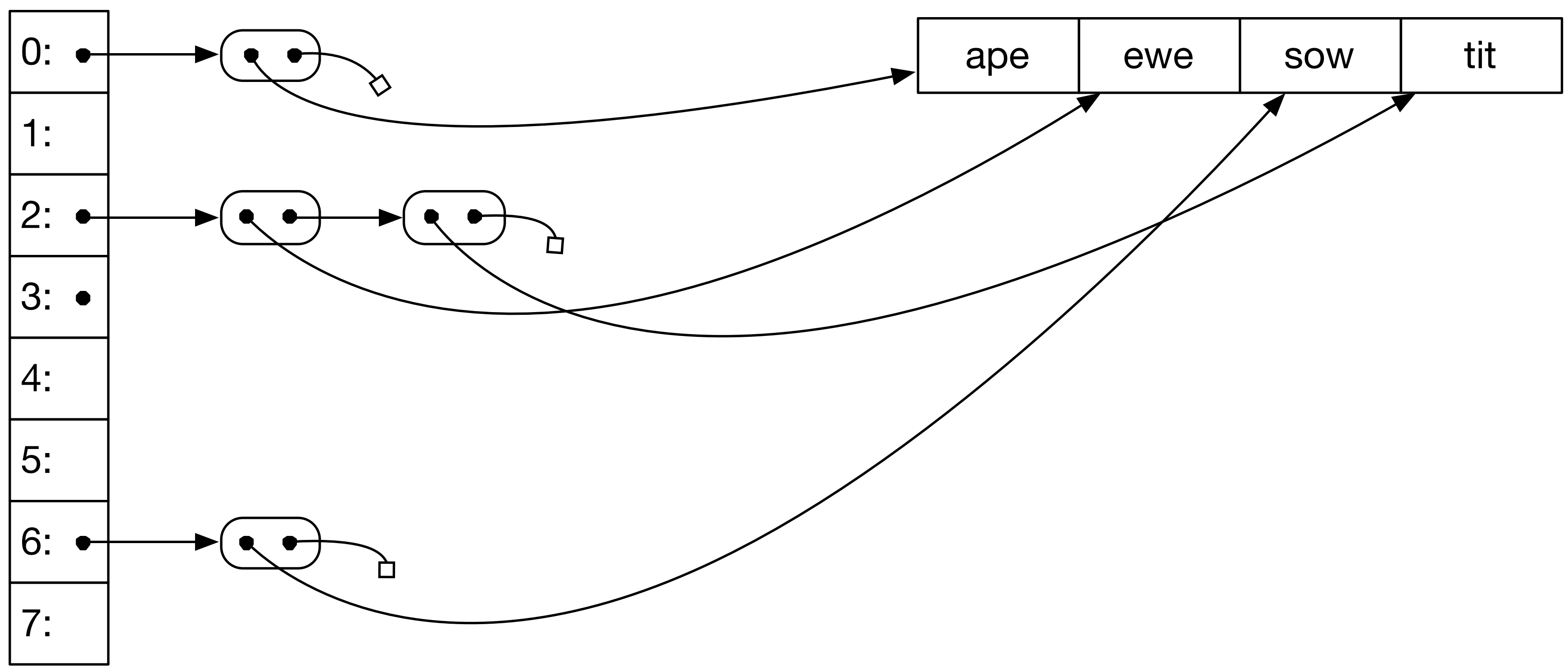
Hashing

- Linear probing leads to convoys:
  - Occupied cells tend to coalesce
- Quadratic probing is better, but might perform worse with long cache lines
- Large number of better versions are used:
  - Passbits
  - Cuckoo hashing
    - Uses two hash functions
  - Robin Hood hashing …

Hashing

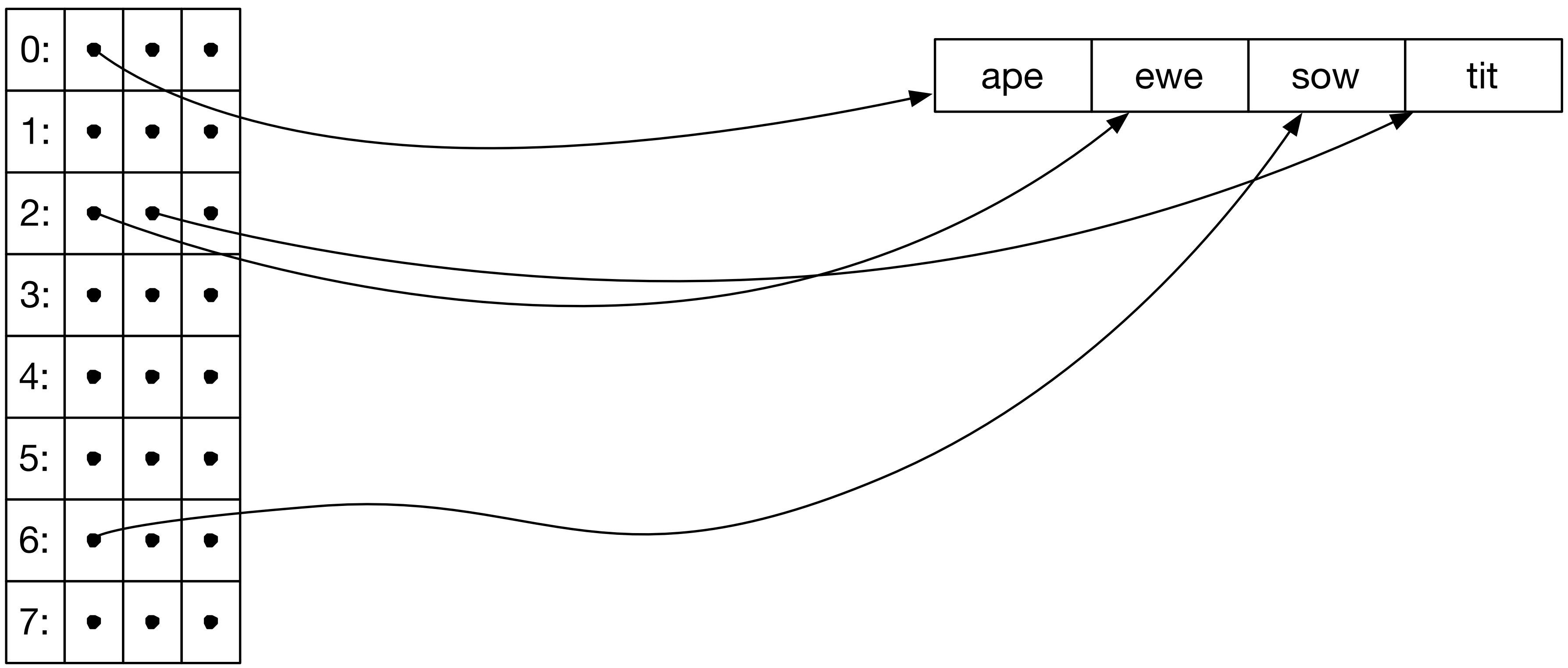
- Chaining
  - Keep data mapped to a location in a “bucket”
    - Can implement the bucket in several ways
      - Linked List

Hashing



Chaining Example with linked lists

Hashing Example



Chaining Example with an array of po
(with overflow pointer if necessary)

Hashing Example

0:	ape	null	null
1:	null	null	null
2:	ewe	tit	null
3:	null	null	null
4:	null	null	null
5:	null	null	null
6:	sow	null	null
7:	null	null	null

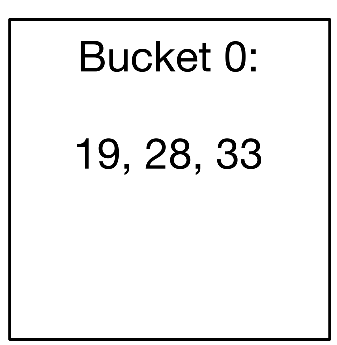
```
Chaining with fixed buckets
Each bucket has two slots and a poi
to an overflow bucket
```

Hashing

- Extensible Hashing:
  - Load factor α = Space Used / Space Provided
  - Load factor determines performance
  - Idea of extensible hashing:
    - Gracefully add more capacity to a growing hash table

Linear Hashing

- Assume a hash function that creates a large string of bits
  - We start using these bits as we extend the address space
  - Start out with a single bucket, Bucket 0
  - All items are located in Bucket 0



Items with keys 19, 28, 33

Linear Hashing

- Eventually, this bucket will overflow
  - E.g. if the load factor is more than 2
  - Bucket 0 splits
  - All items in Bucket 0 are rehashed:
    - Use the last bit in order to determine whether the item goes into Bucket 0 or Bucket 1
    - Address is $h_1(c) = c \pmod 2$

Linear Hashing

- After the split, the hash table has two buckets:

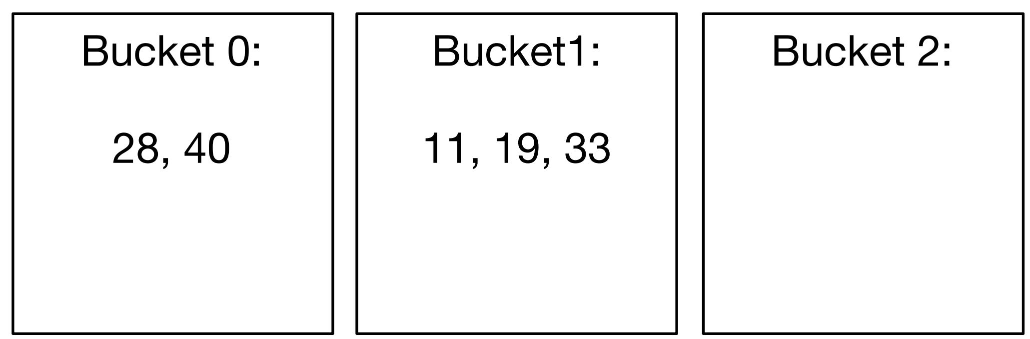
Bucket 0:	Bucket1:
28	19, 33

- After more insertions, the load factor again exceeds 2

Bucket 0:	Bucket1:
28, 40	11, 19, 33

Linear Hashing

- Again, the bucket splits.
  - But it has to be Bucket 0



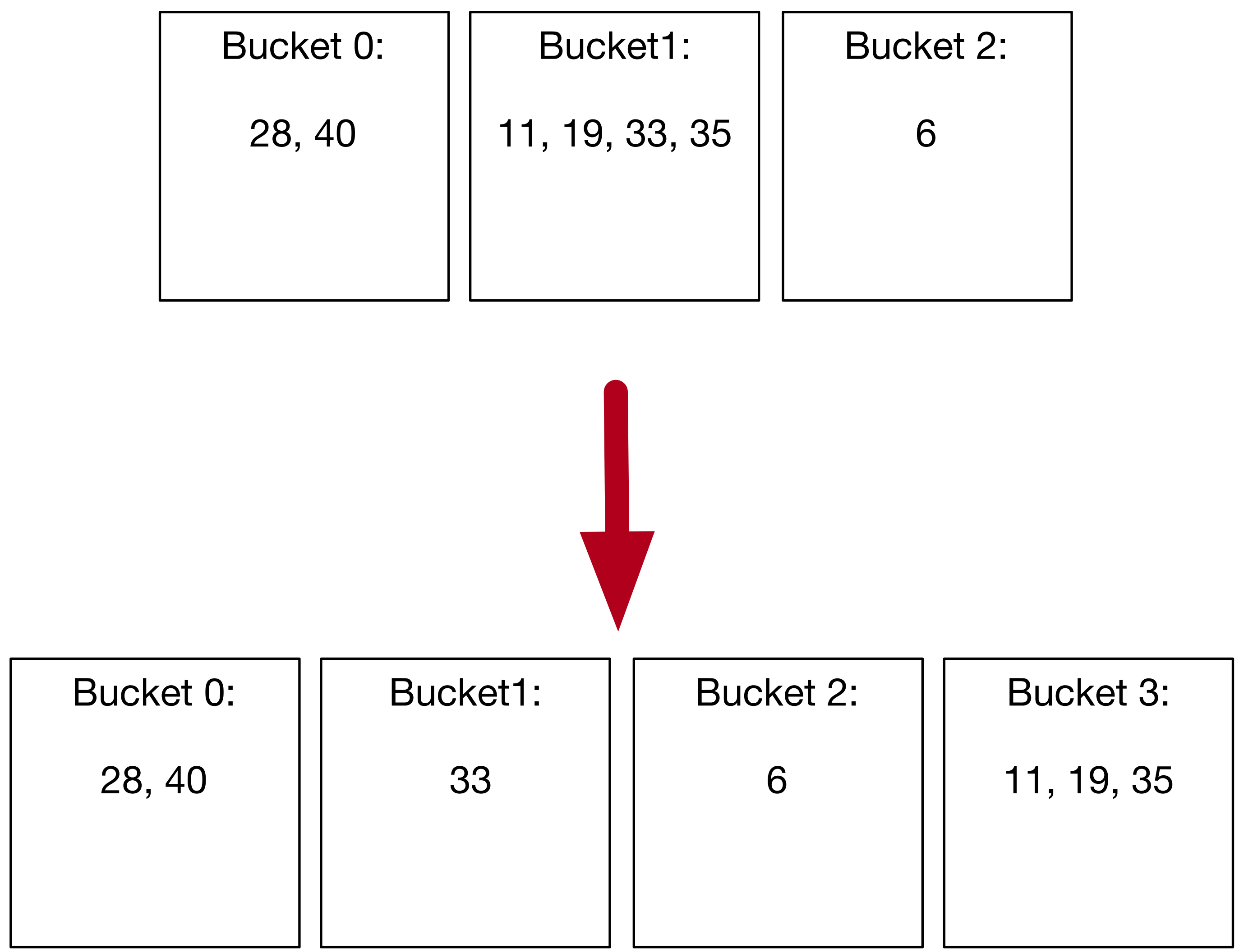
  - For the rehashing, we now use two bits, i.e.

$$h_2(c) = c \pmod 4$$

    - But only for those items in Bucket 0

Linear Hashing

- After some more insertions, Bucket 1 will split



Linear Hashing

- The state of a linear hash table is described by the number N of buckets
    - The level l is the number of bits that are being used to calculate the hash
    - The split pointer s points to the next bucket to be split
    - The relationship is

$$N = 2^l + s$$

        - This is unique, since always $s < 2^l$

Linear Hashing

- Addressing function
  - The address of an item with key c is calculated by

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

  - This reflects the fact that we use more bits for buckets that are already split

Linear Hashtable Evolution

$N = 1 = 2^0 + 0$

```
Number of buckets: 1
Split pointer: 0
Level: 0
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
     a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:

19, 28, 33

Linear Hashtable Evolution

$N = 2 = 2^1 + 0$

```
Number of buckets: 2
Split pointer: 0
Level: 1
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
     a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:
28	19, 33

```
Add items with hashes 40 and 11
This gives an overflow and we split Bucket 0
```

Linear Hashtable Evolution

$N = 3 = 2^1 + 1$

```
Number of buckets: 3
Split pointer: 1
Level: 1
```

```
def address(c):
    a = hash(c) % 2**l
    if a < s:
        a = hash(c) % 2**(l+1)
    return a
```

Bucket 0:	Bucket1:
28, 40	11, 19, 33

```
split Bucket 0
Create Bucket 2
Use new hash function on items in Bucket 0
```

Bucket 0:	Bucket1:	Bucket 2:
28, 40	11, 19, 33	

```
No items were moved
```

Linear Hashtable Evolution

$N = 3 = 2^1 + 1$

```
Number of buckets: 3
Split pointer: 1
Level: 1
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:	Bucket 2:
28, 40	11, 19, 33	

Add items 6, 35

Bucket 0:	Bucket1:	Bucket 2:
28, 40	11, 19, 33, 35	6

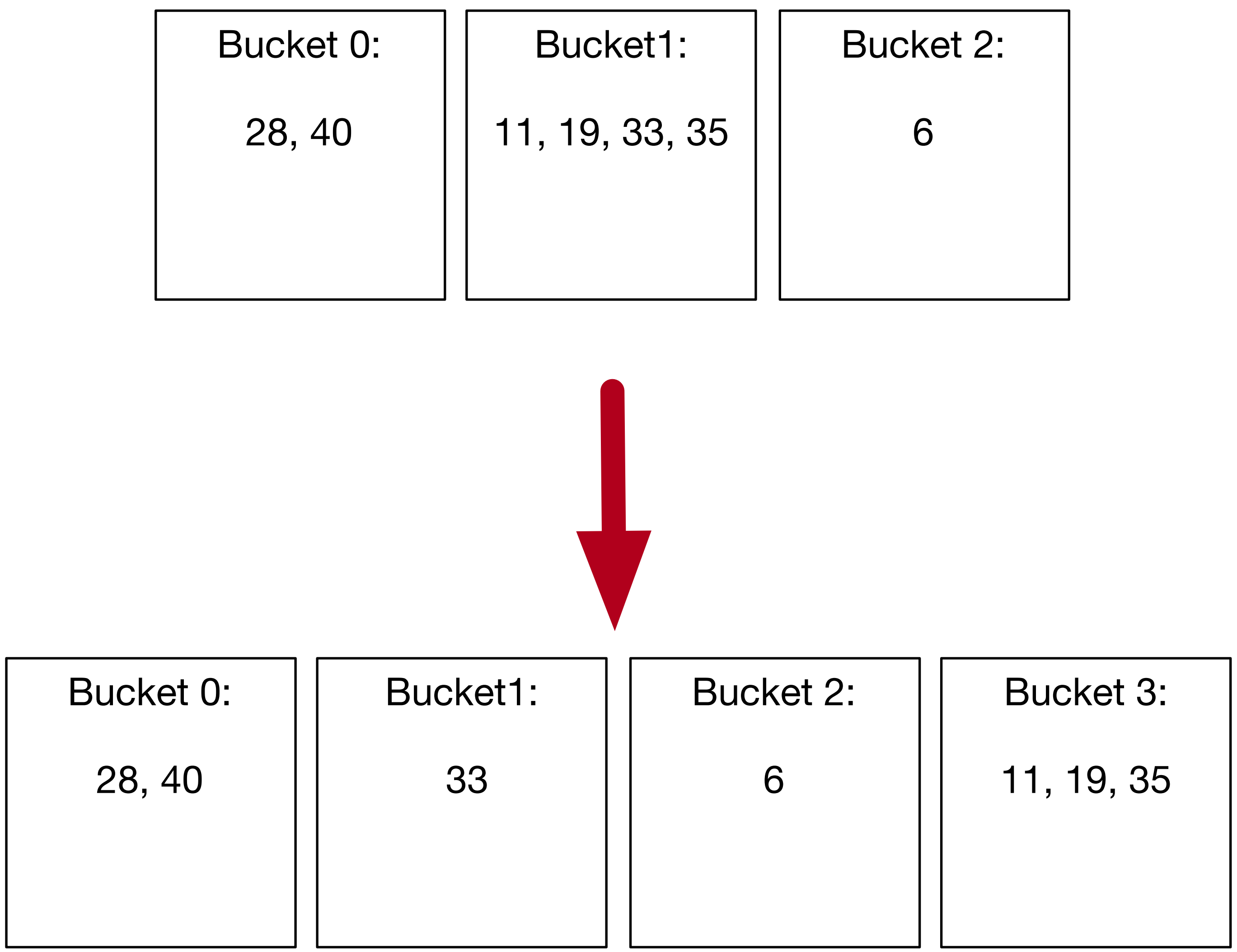
Because of overflow, we split Bucket 1

Linear Hashtable Evolution

$$N = 4 = 2^2 + 0$$

```
Number of buckets: 4
Split pointer: 0
Level: 2
```

```
def address(c):
    a = hash(c) % 2**l
    if a < s:
        a = hash(c) % 2**(l+1)
    return a
```



Linear Hashtable Evolution

$N = 4 = 2^2 + 0$

```
Number of buckets: 4
Split pointer: 0
Level: 2
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:
28, 40	33	6	11, 19, 35

Now add keys 8, 49

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:
28, 40, 8	33, 49	6	11, 19, 35

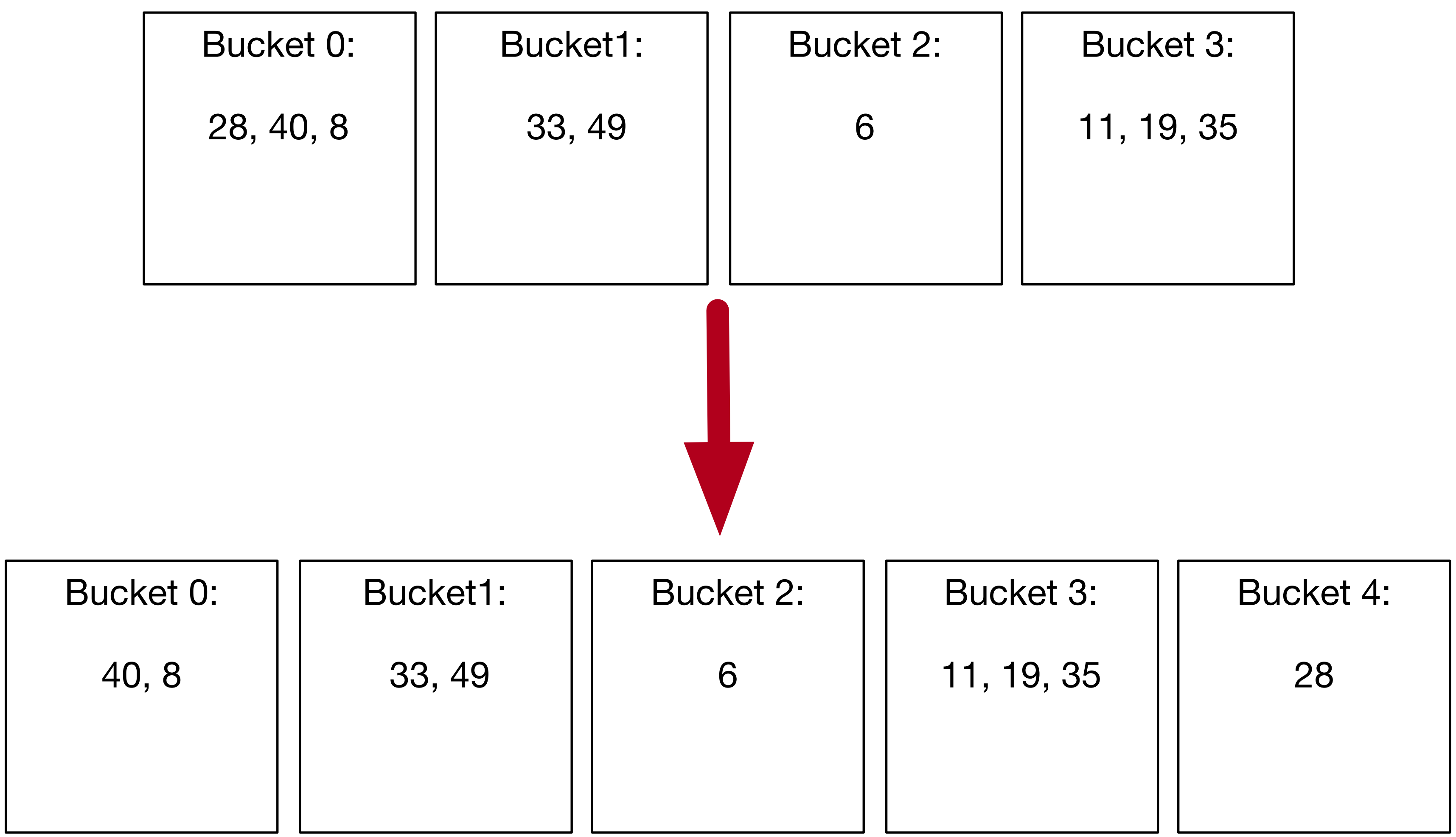
Creates an overflow!
Need to split!

Linear Hashtable Evolution

$$N = 5 = 2^2 + 1$$

```
Number of buckets: 1
Split pointer: 1
Level: 2
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```




```
Create Bucket 4.
Rehash Bucket 0.
```

Linear Hashtable Evolution

$N = 5 = 2^2 + 1$

```
Number of buckets: 5
Split pointer: 1
Level: 2
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:
40, 8	33, 49	6	11, 19, 35	28

Add keys 9, 42

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:
40, 8	9, 33, 49	6, 42	11, 19, 35	28

Creates an overflow!
Need to split!

Linear Hashtable Evolution

$N = 6 = 2^2 + 2$

```
Number of buckets: 1
Split pointer: 2
Level: 2
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:
40, 8	9, 33, 49	6, 42	11, 19, 35	28

Split

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:
40, 8	9, 33, 49	6, 42	11, 19, 35	28	

No item actually moved, but average load factor is now again under 2.

Linear Hashtable Evolution

$$N = 6 = 2^2 + 2$$

```
Number of buckets: 6
Split pointer: 2
Level: 2
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:
40, 8	9, 33, 49	6, 42	11, 19, 35	28	

`add 5,10`

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:
40, 8	9, 33, 49	6, 10, 42	11, 19, 35	28	5

Linear Hashtable Evolution

$N = 7 = 2^2 + 3$

```
Number of buckets: 7
Split pointer: 3
Level: 2
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:
40, 8	9, 33, 49	6, 10, 42	11, 19, 35	28	5

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:
40, 8	9, 33, 49	10, 42	11, 19, 35	28	5	6

Linear Hashtable Evolution

$N = 7 = 2^2 + 3$

```
Number of buckets: 7
Split pointer: 3
Level: 2
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
     a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:
40, 8	9, 33, 49	10, 42	11, 19, 35	28	5	6

`add 92, 74`

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:
40, 8	9, 33, 49	10, 42, 74	11, 19, 35	28, 92	5	6

Linear Hashtable Evolution

$$N = 8 = 2^3 + 0$$

```
Number of buckets: 8
Split pointer: 0
Level: 3
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:
40, 8	9, 33, 49	10, 42, 74	11, 19, 35	28, 92	5	6

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:	Bucket 7:
40, 8	9, 33, 49	10, 42, 74	11, 19, 35	28, 92	5	6	

Linear Hashtable Evolution

$N = 8 = 2^3 + 0$

```
Number of buckets: 8
Split pointer: 0
Level: 3
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
      a = hash(c) % 2**(l+1)
   return a
```

add 13, 54

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:	Bucket 7:
40, 8	9, 33, 49	10, 42, 74	11, 19, 35	28, 92	5	6	

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:	Bucket 7:
	9, 33, 49	10, 42, 74	11, 19, 35	28, 92	5, 13	6, 54	

Linear Hashtable Evolution

$$N = 9 = 2^3 + 1$$

```
Number of buckets: 9
Split pointer: 1
Level: 3
```

```
def address(c):
    a = hash(c) % 2**l
    if a < s:
        a = hash(c) % 2**(l+1)
    return a
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:	Bucket 7:	
	9, 33, 49	10, 42, 74	11, 19, 35	28, 92	5, 13	6, 54		

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:	Bucket 7:	Bucket 8:
	9, 33, 49	10, 42, 74	11, 19, 35	28, 92	5, 13	6, 54		40, 8

Linear Hashtable Evolution

$N = 9 = 2^3 + 1$

```
Number of buckets: 9
Split pointer: 1
Level: 3
```

```
def address(c):
   a = hash(c) % 2**l
   if a < s:
     a = hash(c) % 2**(l+1)
   return a
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:	Bucket 7:	Bucket 8:
	9, 33, 49	10, 42, 74	11, 19, 35	28, 92	5, 13	6, 54		40, 8

add 1, 81

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:	Bucket 7:	Bucket 8:
	1, 9, 33, 49, 81	10, 42, 74	11, 19, 35	28, 92	5, 13	6, 54		40, 8

Linear Hashtable Evolution

$N = 10 = 2^3 + 2$

```
Number of buckets: 10
Split pointer: 2
Level: 3
```

```
def address(c):
    a = hash(c) % 2**l
    if a < s:
        a = hash(c) % 2**(l+1)
    return a
```

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:	Bucket 7:	Bucket 8:	Bucket 9:
	1, 33, 49, 81	10, 42, 74	11, 19, 35, 67, 99	28, 92	5, 13	6, 54	39	40, 8	9

Bucket 0:	Bucket1:	Bucket 2:	Bucket 3:	Bucket 4:	Bucket 5:	Bucket 6:	Bucket 7:	Bucket 8:	Bucket 9:	Bucket 10:
	1, 33, 49, 81		11, 19, 35, 67, 99	28, 92	5, 13	6, 54	39	40, 8	9	10, 42, 74

Linear Hashing

- Observations:
  - Buckets split in fixed order
    - 0, 0,1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, …, 15, 0, …
    - Address calculation is modulo 2^l, i.e. the *l* least significant bits
    - Buckets 0, 1, …, s-1 and 2***l, 2****l*+1, … *N*-1 are already split, they have on average half the size of the buckets *s, s*+1, …, 2***l*.

Linear Hashing

- Observations:
    - An overflowing bucket is not necessarily split immediately
    - Sometimes, a split leaves all keys in the splitting bucket or moves them all to the new bucket
- On average, a bucket will have α items in them