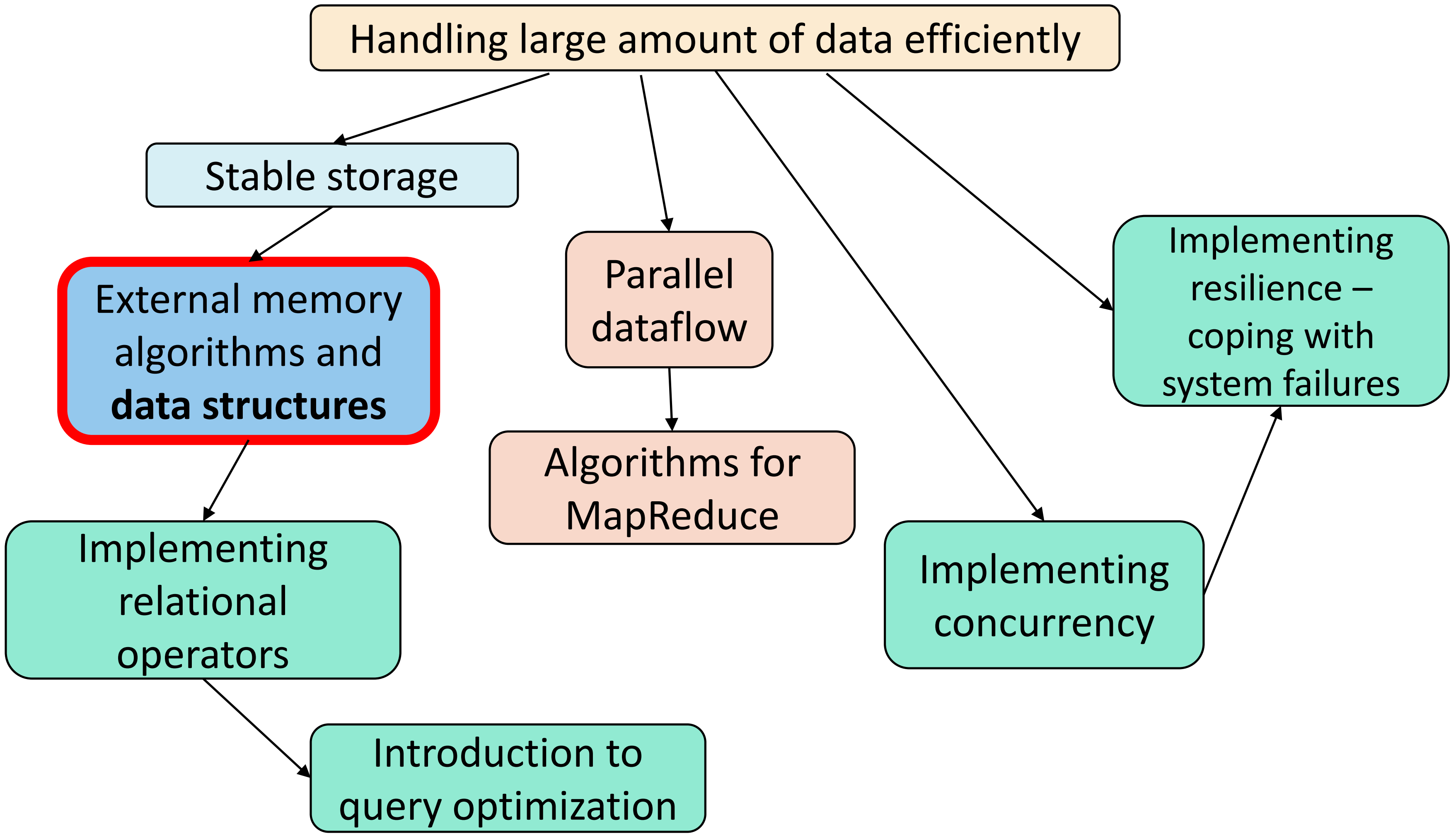
Roadmap


Handling large amount of data efficiently
Stable storage
External memory algorithms and **data structures**
Implementing relational operators
Introduction to query optimization
Parallel dataflow
Algorithms for MapReduce
Implementing concurrency
Implementing resilience – coping with system failures

Lecture 02.06

Dynamic indexes

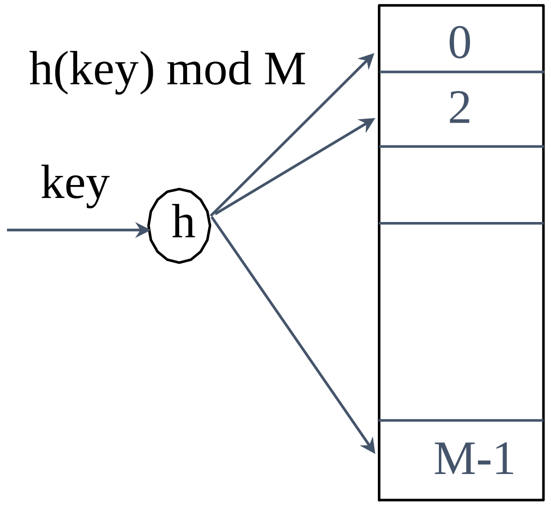
To allow efficient modifications, we need a dynamic data structure, which will guarantee efficient operations in any case

2 main data structures:

- Dynamic Trees
- Dynamic Hashes

By Marina Barsky
Winter 2017, University of Toronto

External-memory hashing

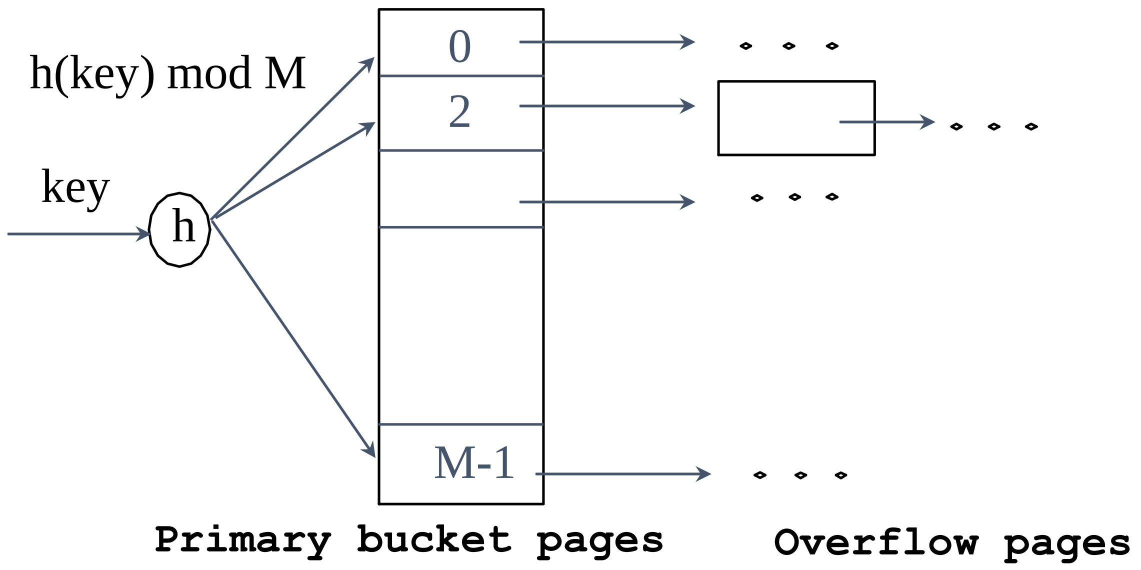


Pages = blocks = buckets

- Buckets contain *data entries*.
- Hash function works on *search key k* of record *r.* Must distribute values over range 0 ... *M*-1.
  - **h**(*k*) = (a * *k* + b) usually works well.
  - a and b are constants; lots known about how to tune **h**.

Static Hashing (Hash-based file organization)

- # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- **h**(*k*) mod M = bucket to which data entry with key *k* belongs. (M = # of buckets)



Static Hashing: increasing number of buckets

- Long overflow chains can develop and degrade performance.
- Efficiency is highest when

  #data entries < (#buckets × #(data entries/bucket))

- If file grows, we need a dynamic method to maintain the above relationship.
  - ***Extensible Hashing***: double the number of buckets when needed.
  - ***Linear hashing***: add one more bucket to increase hash capacity.

Extendible hashing: main idea

- Situation: Bucket (primary page) becomes full. Why not re-organize file by *doubling* # of buckets?
  - Reading and writing all pages is expensive!
  - *Idea*: Use *directory of pointers to buckets*, double # of buckets by *doubling the directory,* splitting just the bucket that overflowed!
  - Directory much smaller than file, so doubling it is much cheaper. Only one page of data entries is split.
  - ***No overflow pages*!**
  - Trick lies in how hash function is adjusted!

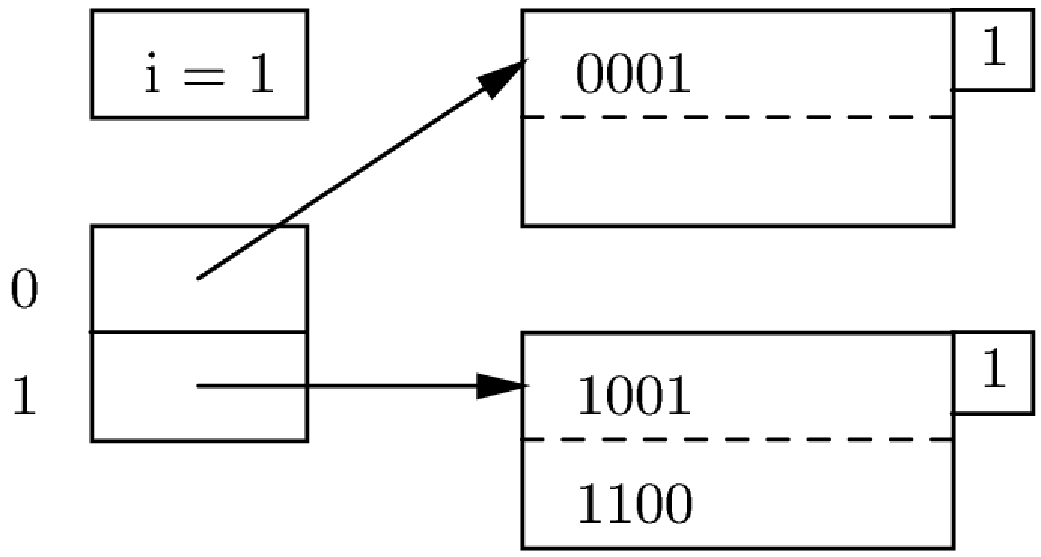
Extendible hashing

- Assume that the hash function ***h(k)*** returns a binary number.
- The first *i* bits* of each binary number will be used as entries in the "directory" which will map these *i* bits to the actual bucket.
- Additionally, *i* is the smallest number such that there are no more data entries with identical first *i* bits that can fit into a single bucket.

---

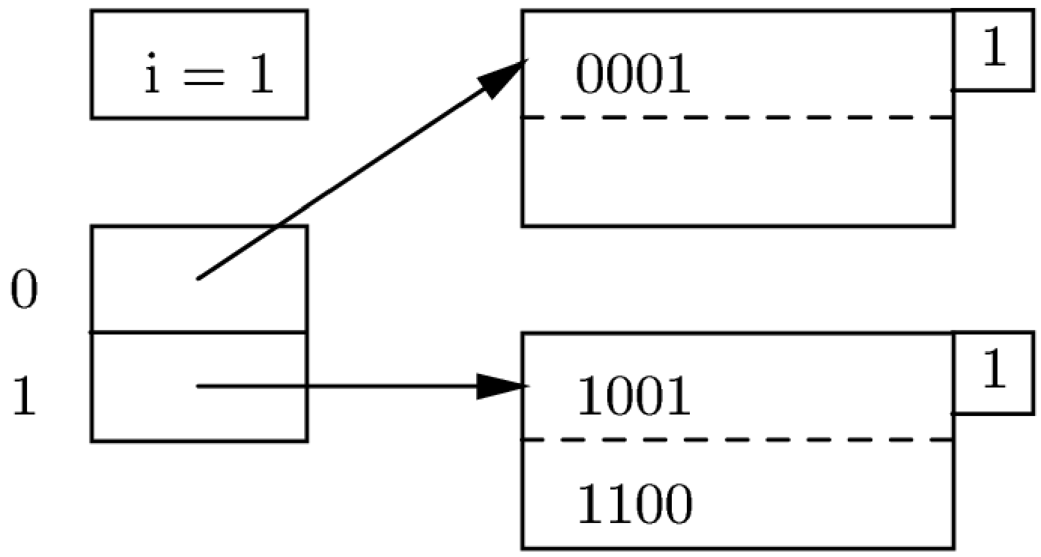
*You can also use the last i bits, by reading them *backwards*. You cannot use the mod function, because in this case you would need to re-distribute all existing keys: for example, key with hash …10 which was in bucket 0, now need to go to bucket 10, while we want to be able to reorganize only a single bucket at a time

Extendible hashing: insertion



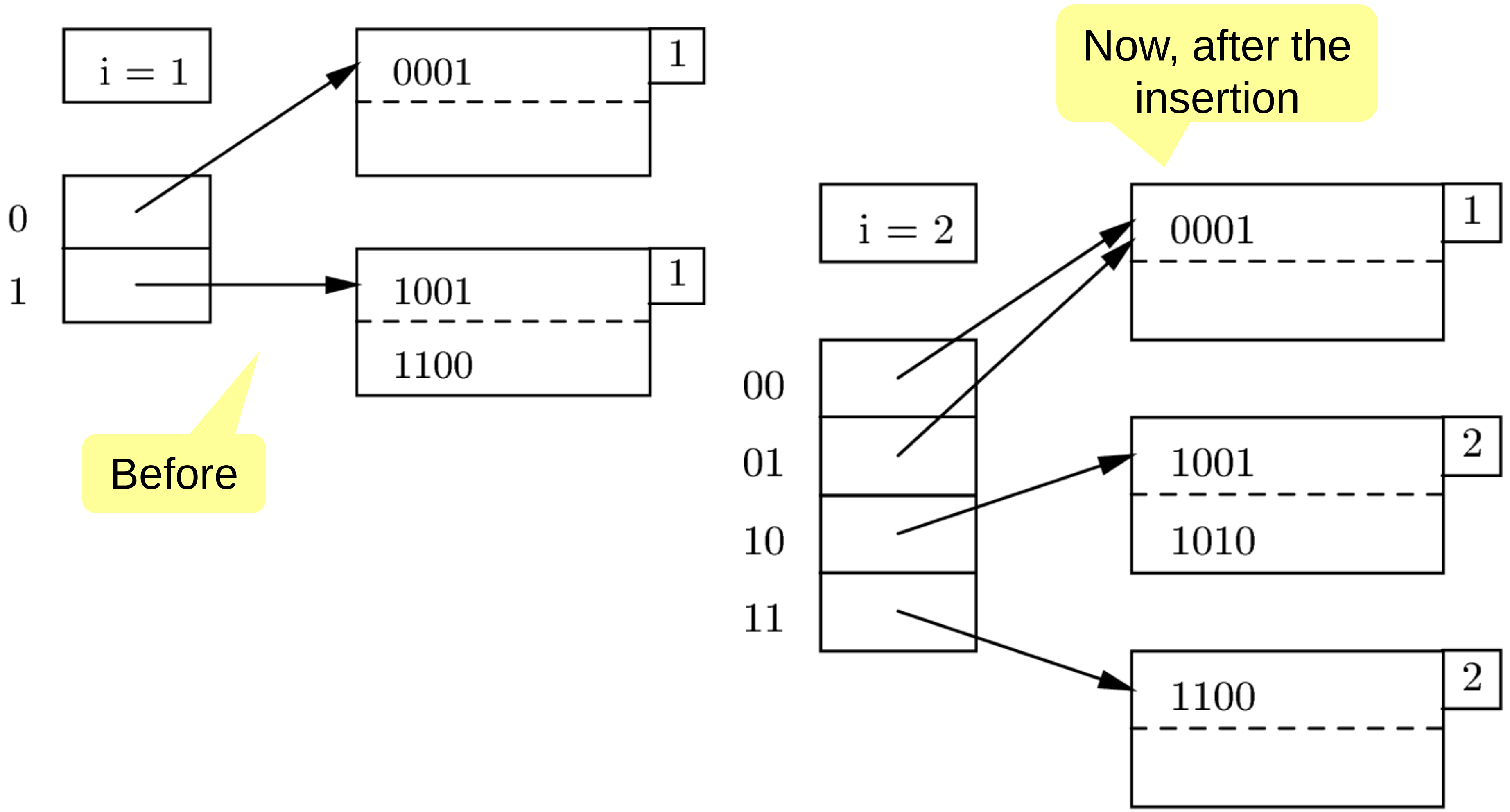
- Directory is array of size 2.
- *Global depth i*=1, i.e. only the first bit of ***h***(*k*) defines placement of a new key
- To find bucket for *k*, take first `*global depth*' # bits of ***h***(*k*)
    - If first bit = 0 it is in bucket pointed to by directory entry 0.
    - If first bit = 1, it is in bucket pointed to by directory entry 1

Extendible hashing: insert record with $h(k) = 1010$



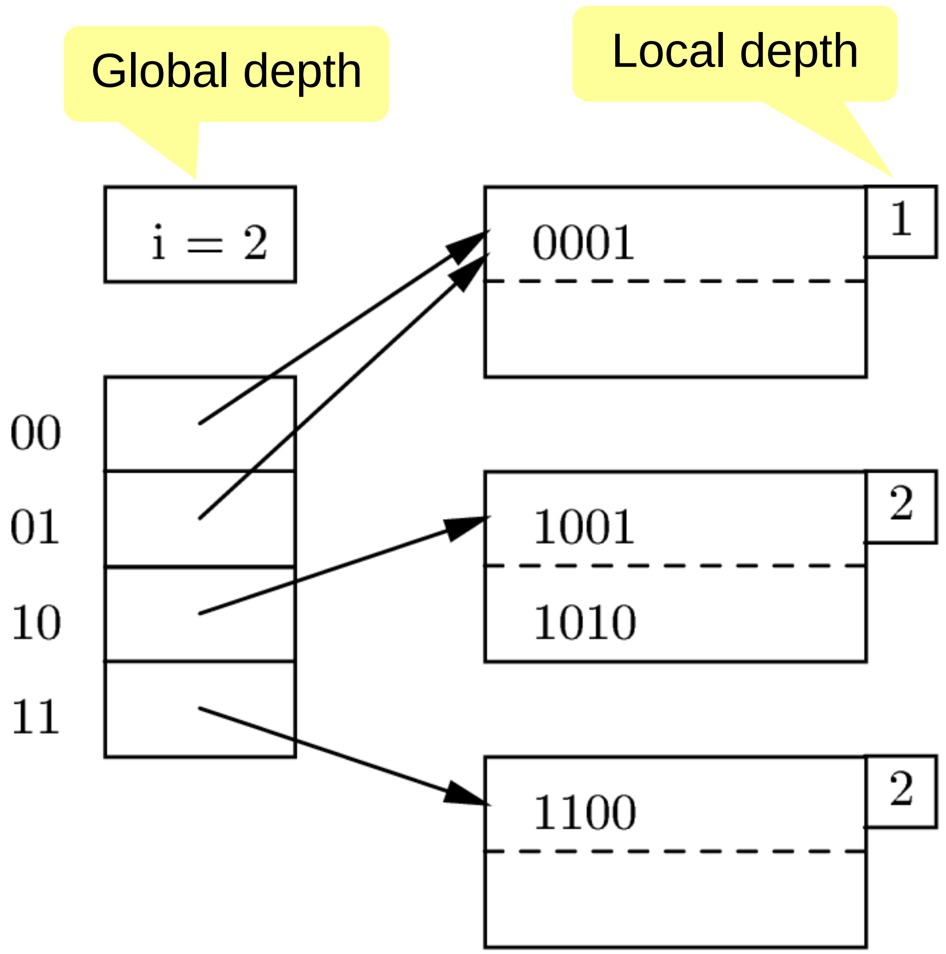
- If bucket is full, *split* it (allocate new page, re-distribute keys according to i+1 bits).
- *If necessary*, double the directory: increment global depth *i*

Extendible hashing: insert record with $h(k) = 1010$

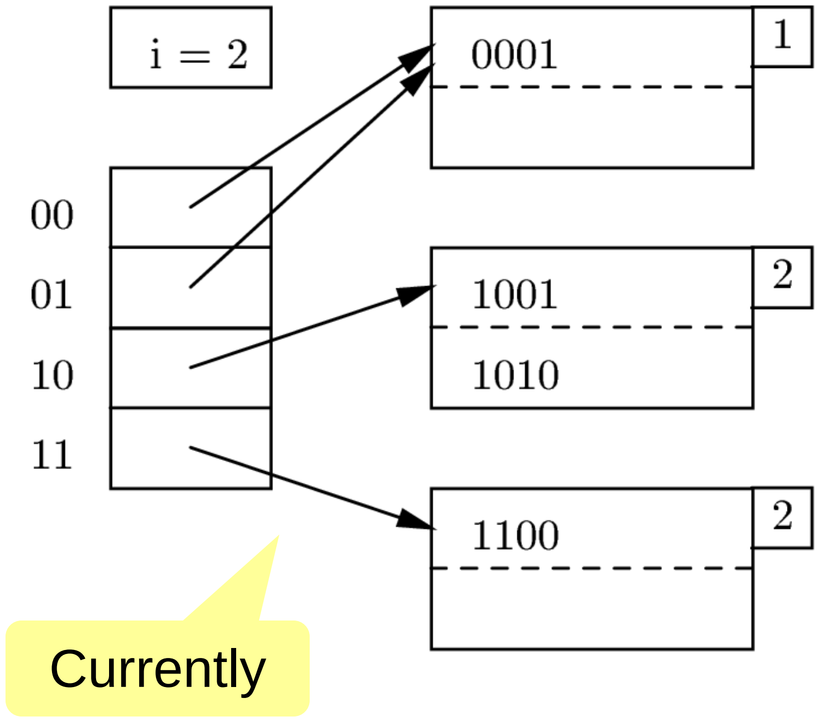


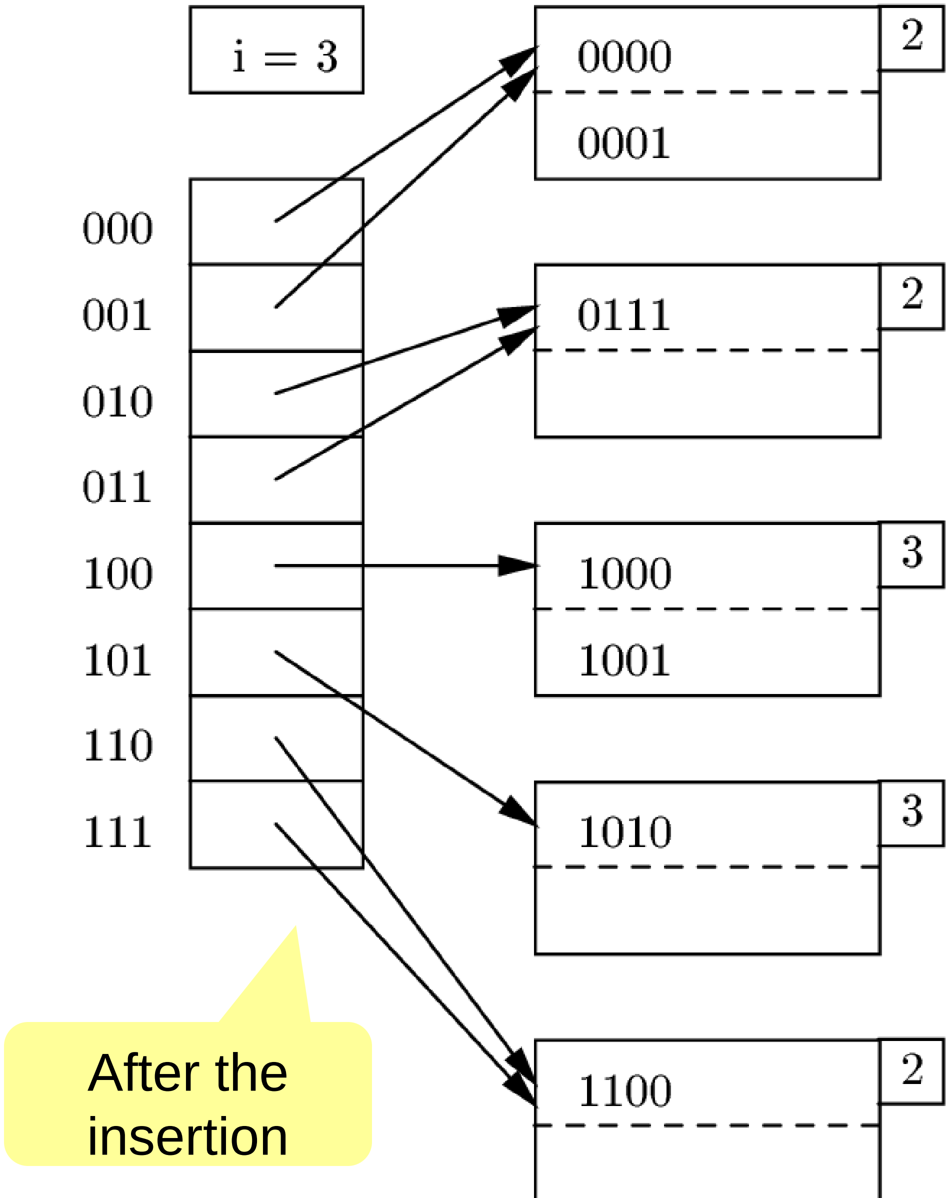
Global depth and local depth

- *Global depth of a directory -* max # of bits needed to tell which bucket an entry belongs to.
- *Local depth of a bucket -* # of bits used to determine if an entry belongs to this bucket.



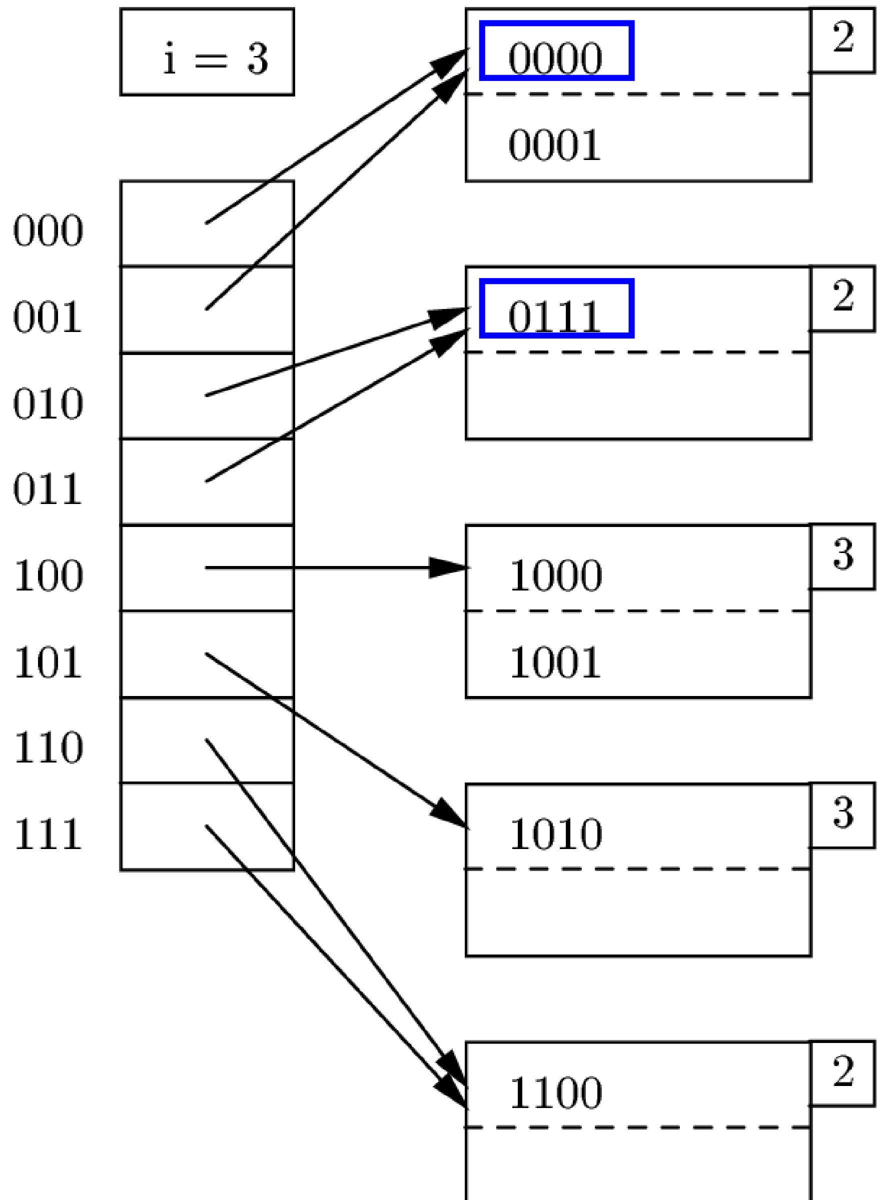
Extendible hashing: insert records with $h(k) = 0000$ $h(k) = 0111$ $h(k) = 1000$





Extendible hashing: insert records with $h(k) = 0000$ $h(k) = 0111$ $h(k) = 1000$

- After insertion of
  $h(K)$=0000; $h(K)$=0111.
  - Bucket for 00 gets split,
  - but *i* stays at 2.

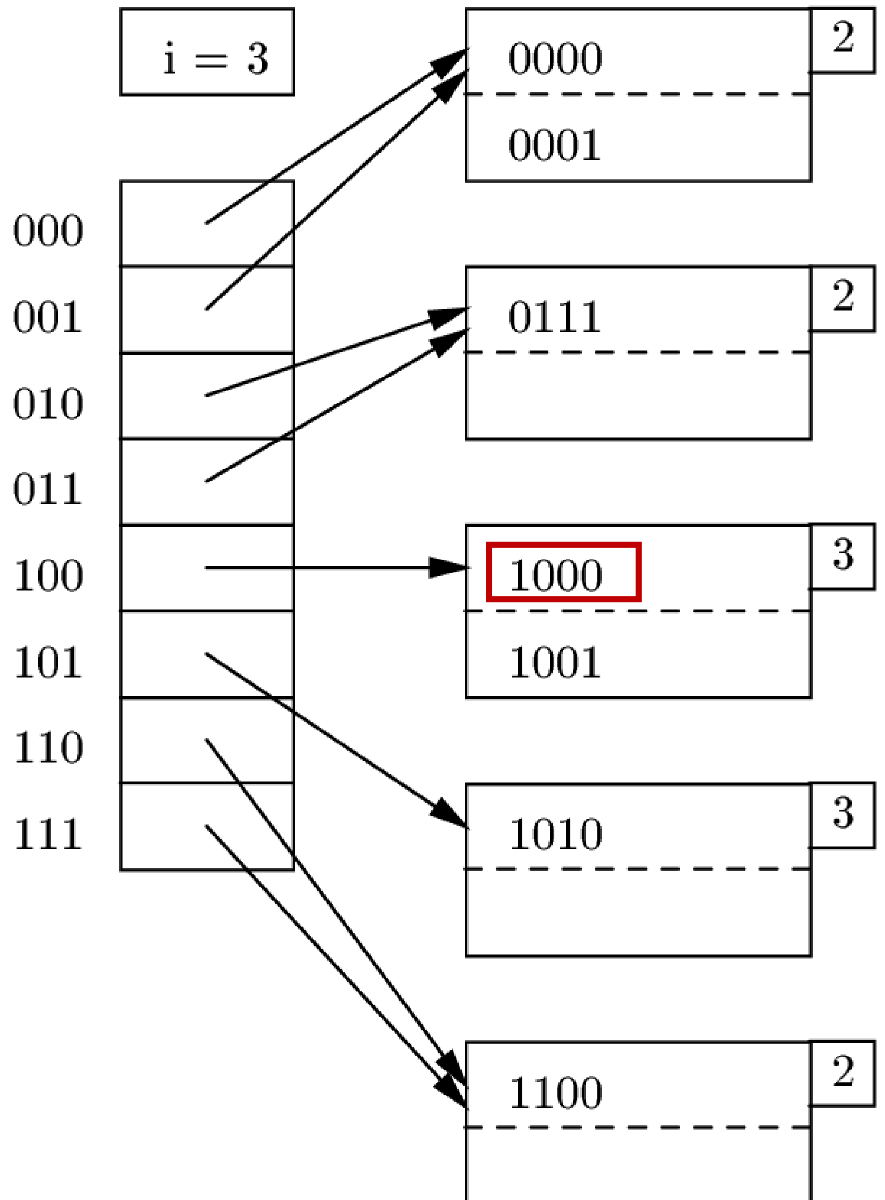


Extendible hashing: insert records with

h(*k*) = 0000

h(*k*) = 0111

h(*k*) = 1000

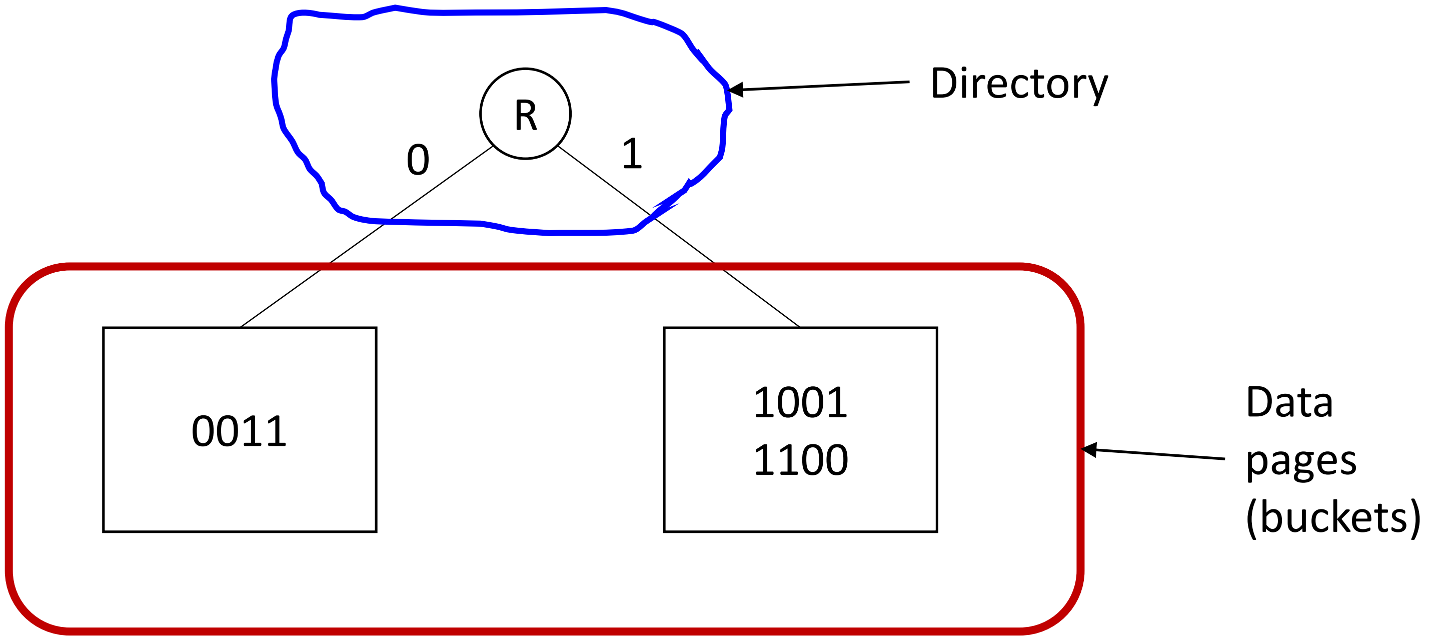


- After insertion of

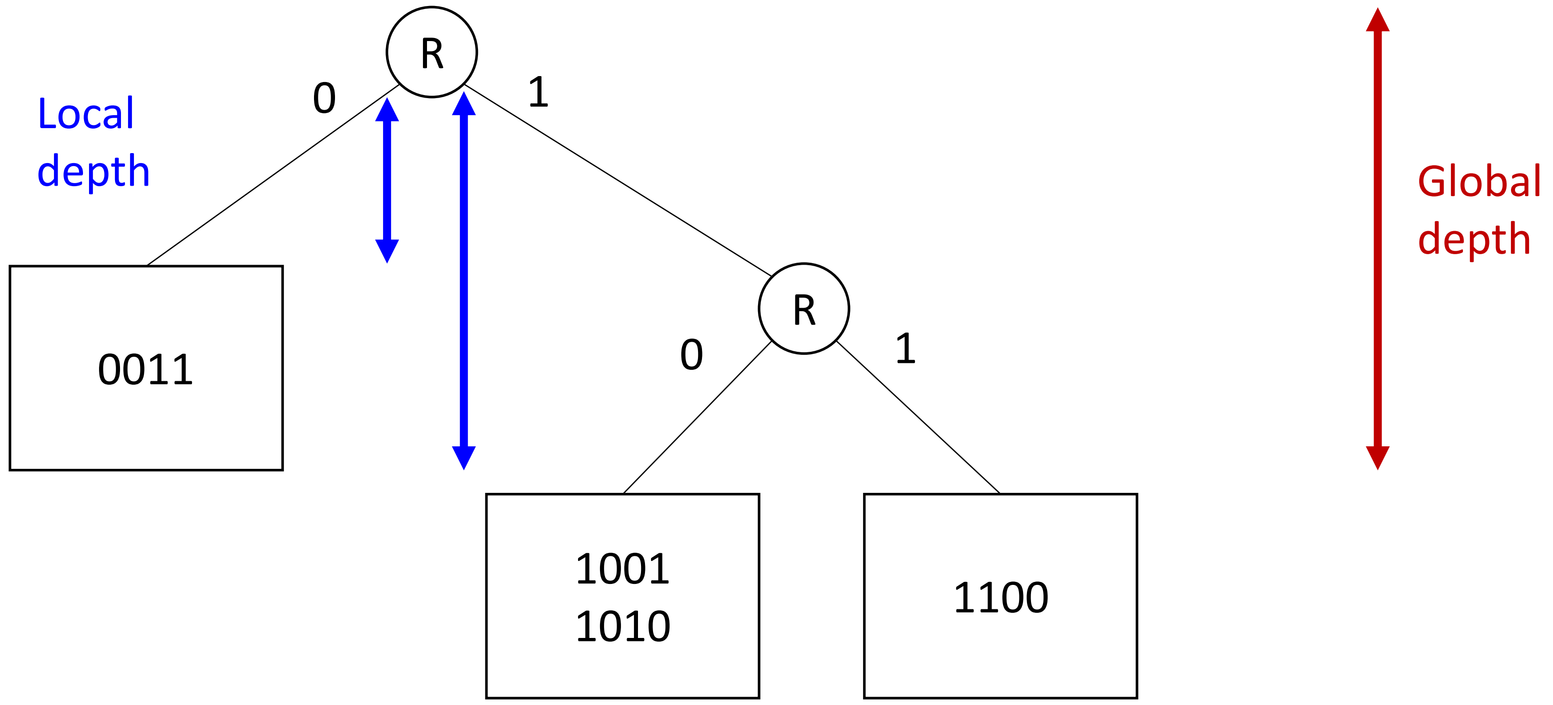
  h(*K*)=0000; *h*(*K*)=0111
  - Bucket for 0... gets split,
  - but *i* stays at 2.
- After insertion of

  h(K) = 1000
  - Overflows bucket for 10...
  - Raise *i* to 3.

Analogy: extending a binary prefix



Analogy: extending a binary prefix



Extendible hash table: lookup

- If directory fits in memory, equality search is answered with one disk access; else two.
- The directory is growing by doubling. All new entries are appended to the end of the directory. Thus, by knowing current global depth, we know exactly where the entry for the first *i* bits of h(k) is.

- The **range search is not supported**: need to scan all the buckets!

Extendible hashing: problems

- Doubling the directory pages can lead to a very large directory.
- There are no overflow pages, that means that we are going to double the directory size until we managed to fit conflicting entries to different buckets.
- Problem with skewed key distributions.
  - E.g. Let 1 block=2 records. Suppose that three records have hash values, which happen to be the same in the first 20 bits.
  - In that case we would have i=20 and one million bucket-array entries, even though we have only 3 records!!

Linear Hashing

- *Idea*: Use a family of hash functions $\mathbf{h}_0$, $\mathbf{h}_1$, $\mathbf{h}_2$, ...
    - $\mathbf{h}_i(k) = \mathbf{h}(k) \bmod(2^i n)$; n = initial # buckets

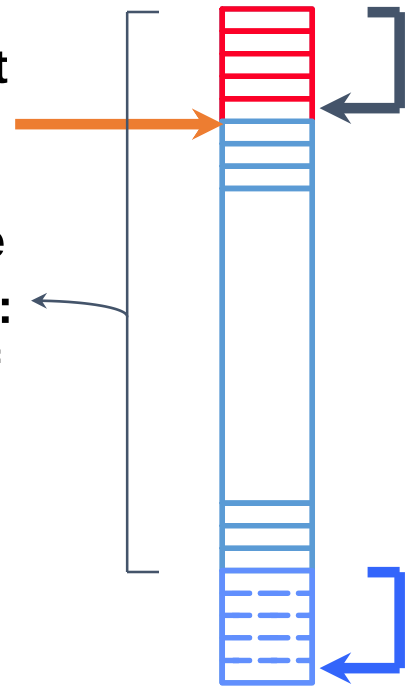
- Directory avoided in LH by using overflow pages, and choosing bucket to split round-robin.

Linear Hashing

- Splitting proceeds in `***rounds***'.
- The bucket to be split is pointed to by ***s***. At each split, ***s*** is incremented.
- Round ends when all N_i initial (for round ***i***) buckets are split.
- At each point in time, buckets 0 to ***s-1*** have been split, all the rest is yet to be split.
- When the round ends, ***s*** is reset to 0, and a new round begins
- Current round number is ***i***.
- **<u>Search:</u>** To find bucket for data entry *k,* find $\mathbf{h}_i(k)$*:*
    - If $\mathbf{h}_i(k) >= \mathbf{s}$, *k* belongs to bucket $\mathbf{h}_i(k)$.
    - Else, apply $\mathbf{h}_{i+1}(k)$ to find out the bucket.

Overview of Linear Hashing

In the middle of a round.



Bucket to be split

s

Buckets that existed at the beginning of this round: this is the range of

h_i

Buckets split in this round: If h_i (search key value) is in this range, must use h_{i+1} (search key value) to decide if entry is in `split image' bucket.

`split image' buckets: created (through splitting of other buckets) in this round

Linear Hashing: insertion

- **Insert**: Find bucket by applying $\mathbf{h}_i$ or $\mathbf{h}_{i+1}$*:*
  - If bucket to insert into is full:
    - Add overflow page and insert data entry.
    - (*Maybe*) Split bucket *s* and increment *s*.
- Can choose any criterion to `trigger’ split.
- Since buckets are split round-robin, long overflow chains don’t develop!
- Doubling of directory in Extendible Hashing is similar; switching of hash functions is *implicit* in how the # of bits examined is increased.

Linear Hashing (LH): insertion example

The sequence of keys to be inserted

1	7	3	8	12	4	11	2	10	13	5

Initial array of buckets

0		
1		

LH: setup

The sequence of keys to be inserted

1	7	3	8	12	4	11	2	10	13	5

Initial array of buckets

0		
1		

M – current number of buckets, initially M = 2
R – max number of data entries in one bucket, R=2
N – total number of data entries, N=0
N/MR – split threshold: if N/MR = 0.75 – split current bucket

s – pointer to the current bucket – to be split next, initially s=0

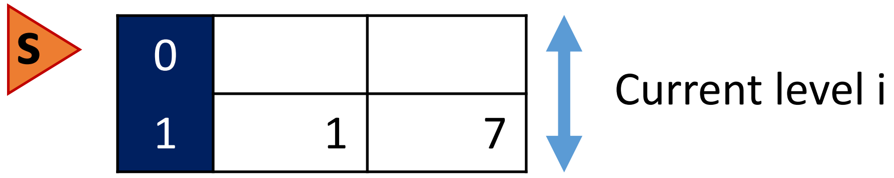
Family of hash functions – depends on the initial number of buckets (2 in this example): h_i (k) = k mod 2^i *2
h_0 (k) = k mod 2, h_1 (k) = k mod 4, …

Current level i=0

Linear hashing: insert 1, 7

The sequence of keys to be inserted

1	7	3	8	12	4	11	2	10	13	5



M = 2, hash capacity = 4

N = 2

i=0

Split when N/hash capacity > 3/4

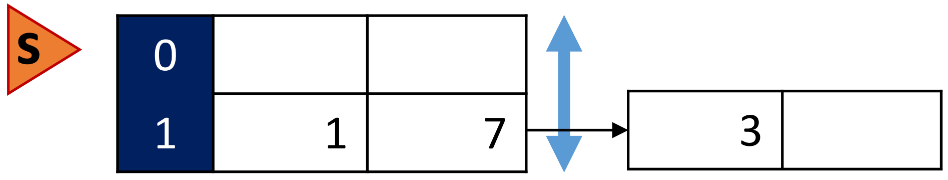
To find where a new record belongs, use h_0 (k) = k mod 2

As far as N/MR <= 0.75, insert into the corresponding bucket

Linear hashing: insert 3

The sequence of keys to be inserted

1	7	3	8	12	4	11	2	10	13	5



M = 2, hash capacity = 4

N = 3

i=0

Split when N/hash capacity > 3/4

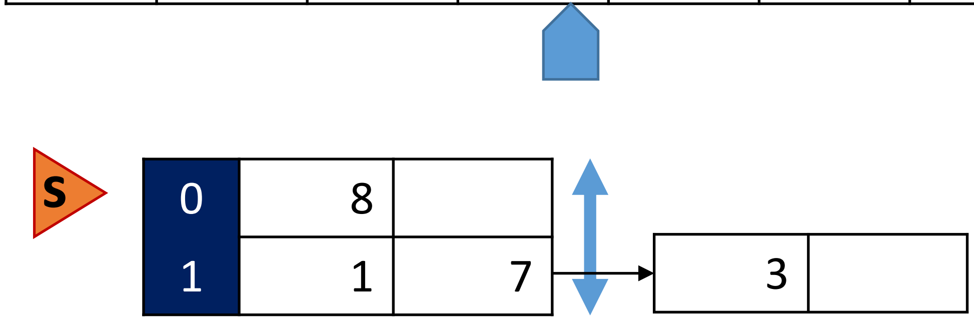
To find where a new record belongs, use h_0 (k) = k mod 2

N/MR = 3/4 <= 0.75, so no split yet. However need an overflow bucket to store 3.
The space of the overflow buckets is not used in the calculation of the split threshold!

Linear hashing: insert 8

The sequence of keys to be inserted

1	7	3	8	12	4	11	2	10	13	5



M = 2, hash capacity = 4

N = 4

i=0

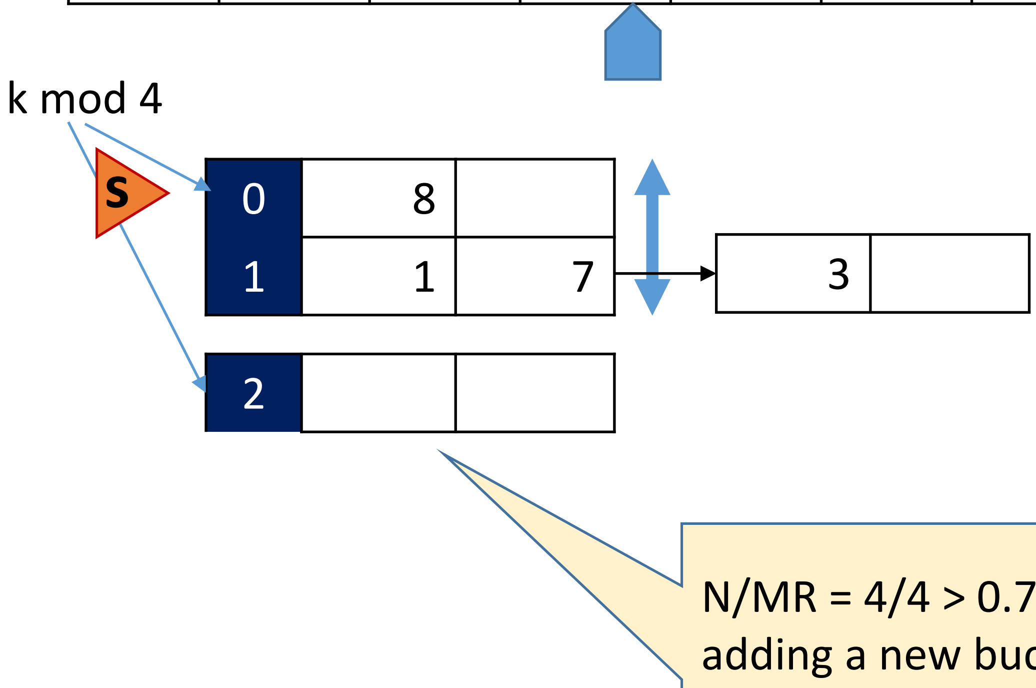
Split when N/hash capacity > 3/4

To find where a new record belongs, use h_0 (k) = k mod 2

Linear hashing: insert 8

The sequence of keys to be inserted

1	7	3	8	12	4	11	2	10	13	5



M = 3, hash capacity = 6

N = 4

i=0

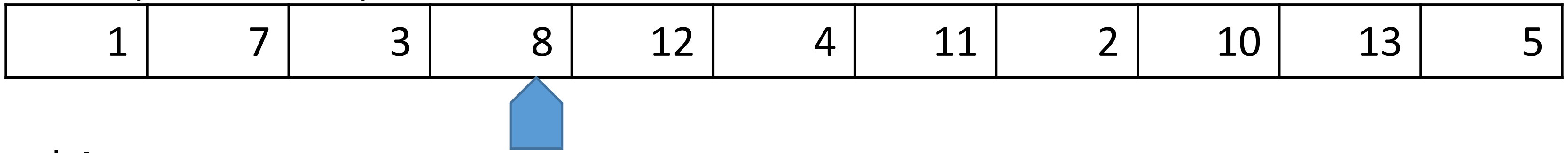
Split when N/hash capacity > 3/4

To find where a new record belongs, use h_0 (k) = k mod 2

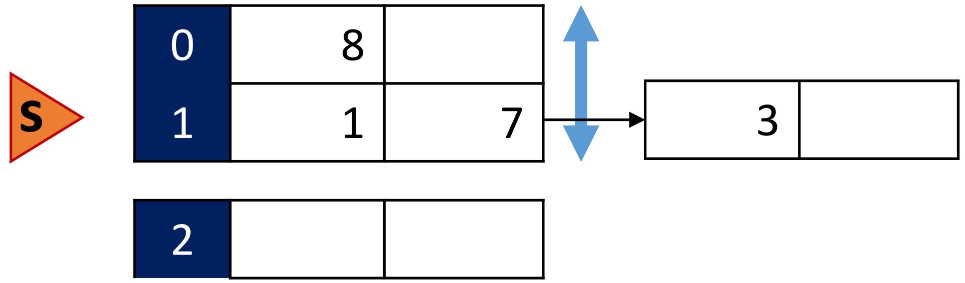
N/MR = 4/4 > 0.75 => split current bucket by adding a new bucket, and rehash keys of bucket s with the next hashing function h_1 (k) = k mod 4. No key re-distribution needed (8 mod 4 =0, so remains in bucket 0)

Linear hashing: insert 8

The sequence of keys to be inserted



k mod 4



M = 3, hash capacity = 6

N = 4

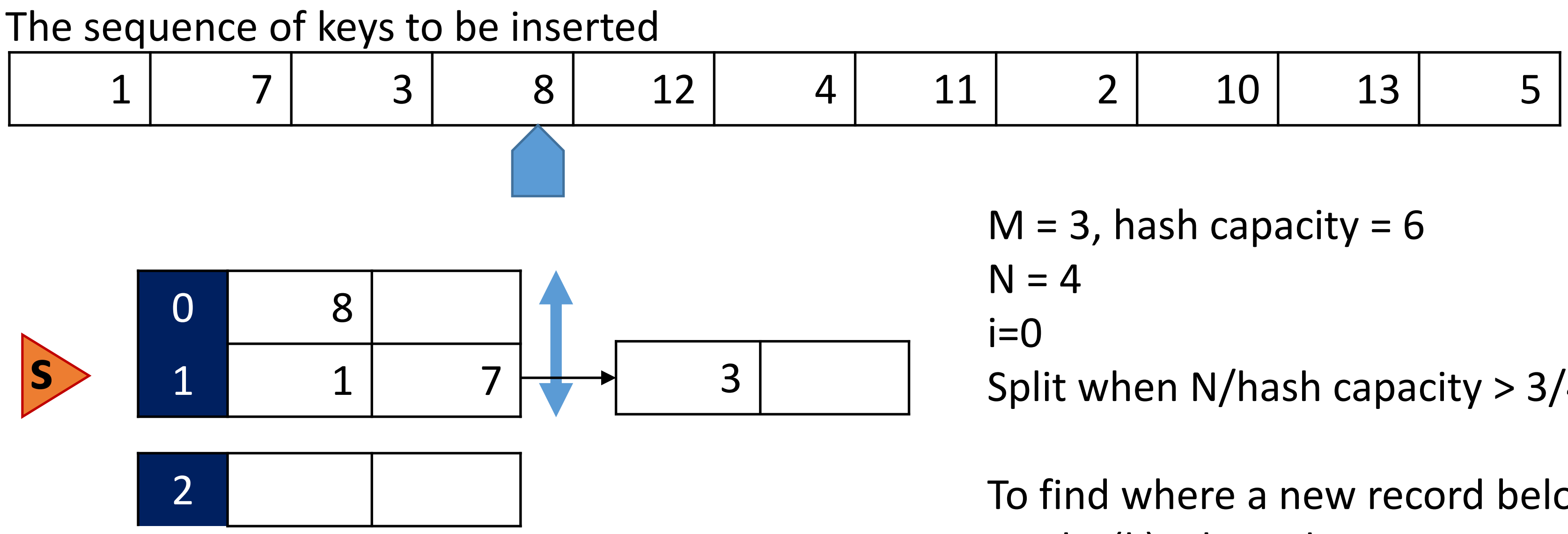
i=0

Split when N/hash capacity > 3/4

To find where a new record belongs, use h_0 (k) = k mod 2

We have split current bucket s. Advance s to the next bucket of the current level

Linear hashing: insert 8



M = 3, hash capacity = 6

N = 4

i=0

Split when N/hash capacity > 3/4

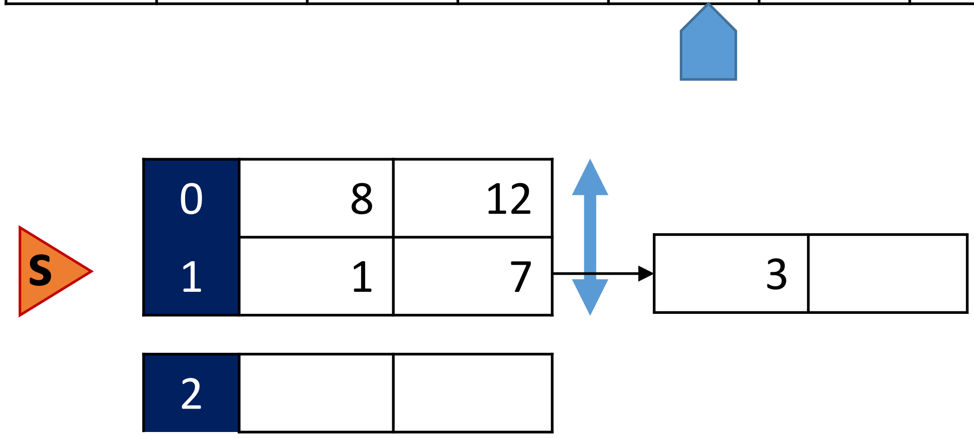
To find where a new record belongs, use h_0 (k) = k mod 2

Now to find the place for a new key, we first use k mod 2. If the result is 0 (above the current position of s), then we use k mod 4.

Linear hashing: insert 12

The sequence of keys to be inserted

1	7	3	8	12	4	11	2	10	13	5



M = 3, hash capacity = 6

N = 5

i=0

Split when N/hash capacity > 3/4

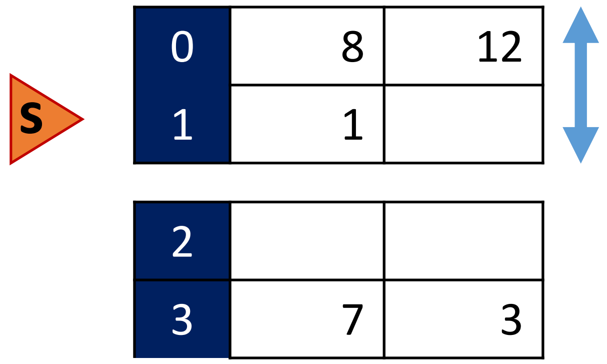
To find where a new record belongs,
use h_0 (k) = k mod 2
If the result is < s,
Use h_1 (k) = k mod 4

12 mod 2 = 0, 0 < s, use h_1= 12 mod 4.
Still belongs to bucket 0.
Check for split threshold:
5/6 > 3/4. We need to split the current bucket

Linear hashing: insert 12

The sequence of keys to be inserted

1	7	3	8	12	4	11	2	10	13	5



M = 4, hash capacity = 8

N = 5

i=0

Split when N/hash capacity > 3/4

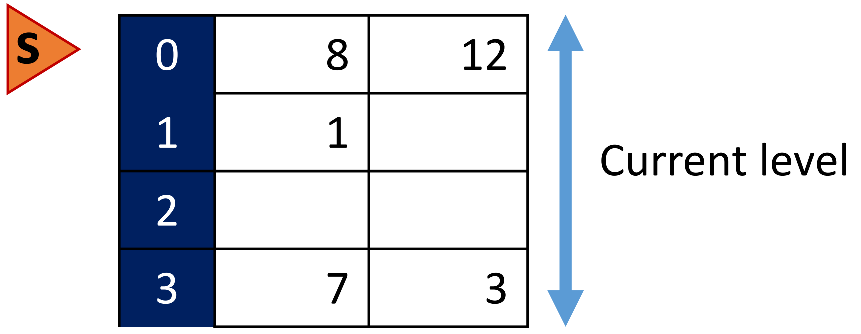
To find where a new record belongs,
use h_0 (k) = k mod 2
If the result is < s,
Use h_1 (k) = k mod 4

We add a new bucket, and re-distribute records from page 1 between 1 and 3, by applying mod 4.
The overflow page is removed and space is reclaimed. 5/8 < 3/4

Linear hashing: insert 12

The sequence of keys to be inserted

1	7	3	8	12	4	11	2	10	13	5



M = 4, hash capacity = 8
N = 5
i=1
Split when N/hash capacity > 3/4

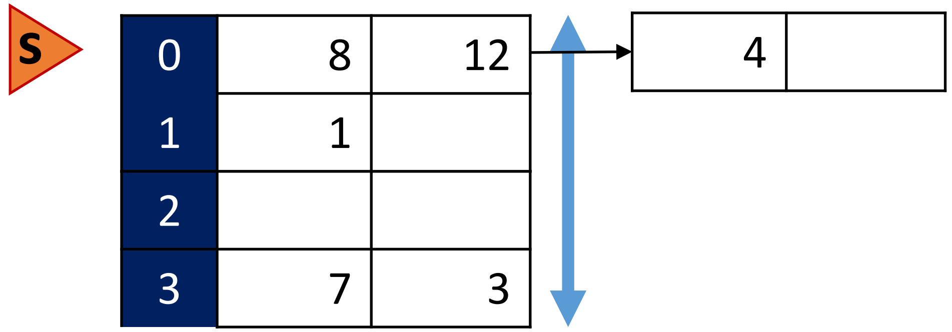
To find where a new record belongs, use h_1 (k) = k mod 4

Because we have finished splitting of a current level i, we reset s to the beginning of a new level i+1.
In the next round, we use mod 4 for all the new keys.

Linear hashing: insert 4

The sequence of keys to be inserted

1	7	3	8	12	4	11	2	10	13	5



M = 4, hash capacity = 8

N = 6

i=1

Split when N/hash capacity > 3/4

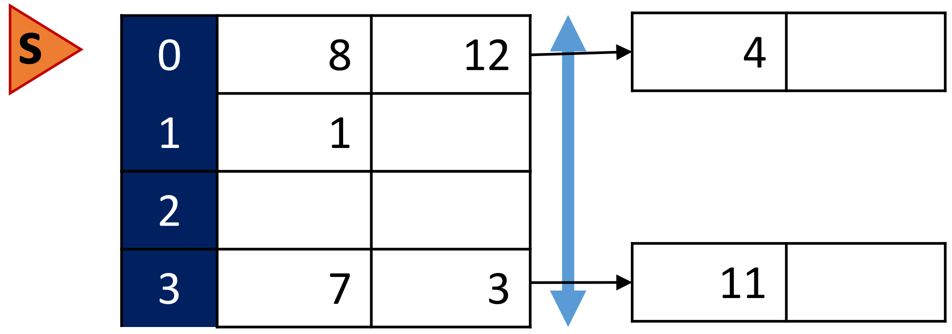
To find where a new record belongs, use h_1 (k) = k mod 4

4 mod 4 hashes to bucket 0, add overflow bucket.
Check for split threshold: 6/8 <= 3/4
- no split needed

Linear hashing: insert 11

The sequence of keys to be inserted

1	7	3	8	12	4	11	2	10	13	5



M = 3, hash capacity = 8

N = 7

i=1

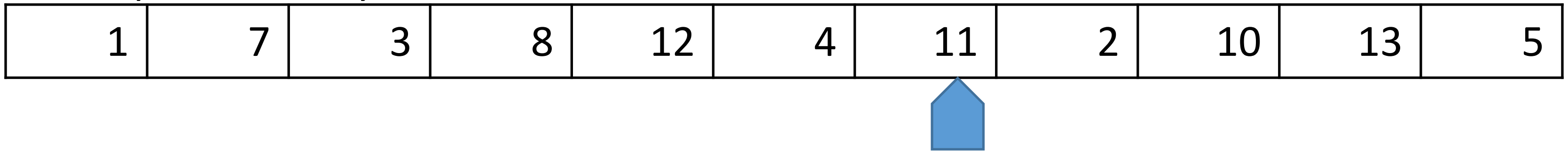
Split when N/hash capacity > 3/4

To find where a new record belongs, use h_1 (k) = k mod 4

11 mod 4 hashes to bucket 3, add overflow bucket.
Check for split threshold: 7/8 > 3/4
- need to split current bucket s

Linear hashing: insert 11

The sequence of keys to be inserted



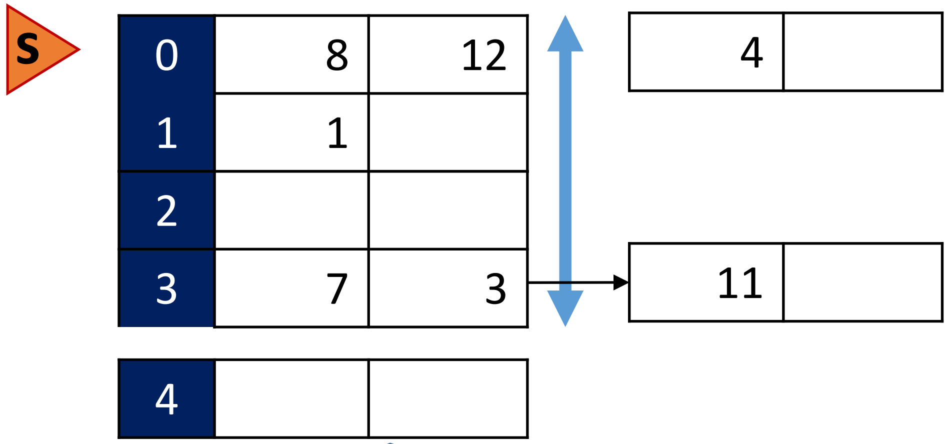
M = 5, hash capacity = 10

N = 7

i=1

Split when N/hash capacity > 3/4

To find where a new record belongs, use h_1 (k) = k mod 4

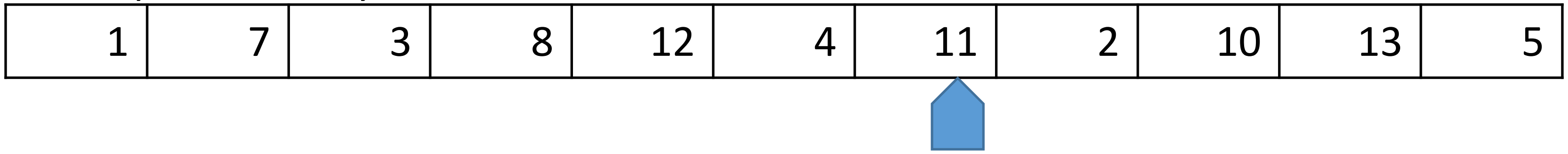


Add a new page.

Linear hashing: insert 11

The sequence of keys to be inserted

1	7	3	8	12	4	11	2	10	13	5



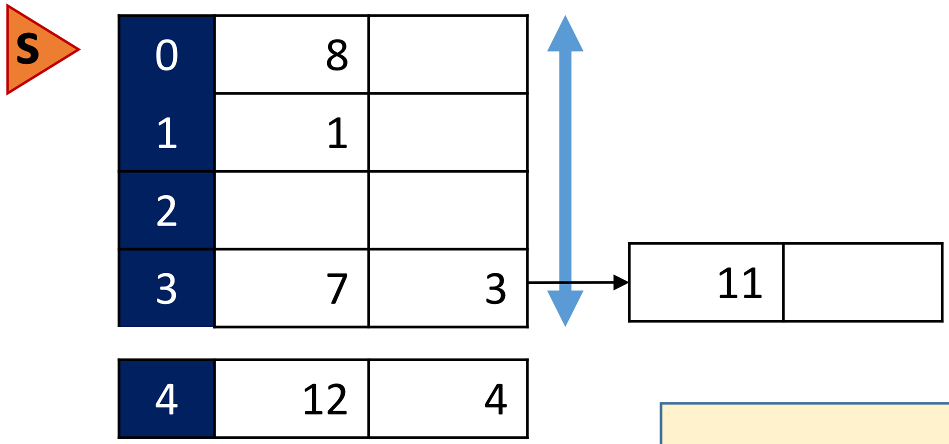
M = 5, hash capacity = 10

N = 7

i=1

Split when N/hash capacity > 3/4

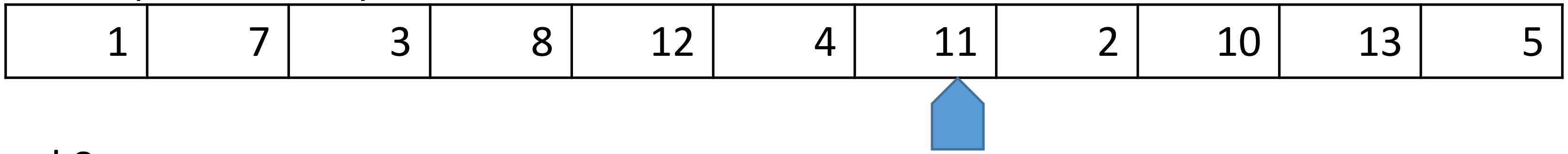
To find where a new record belongs, use h_1 (k) = k mod 4

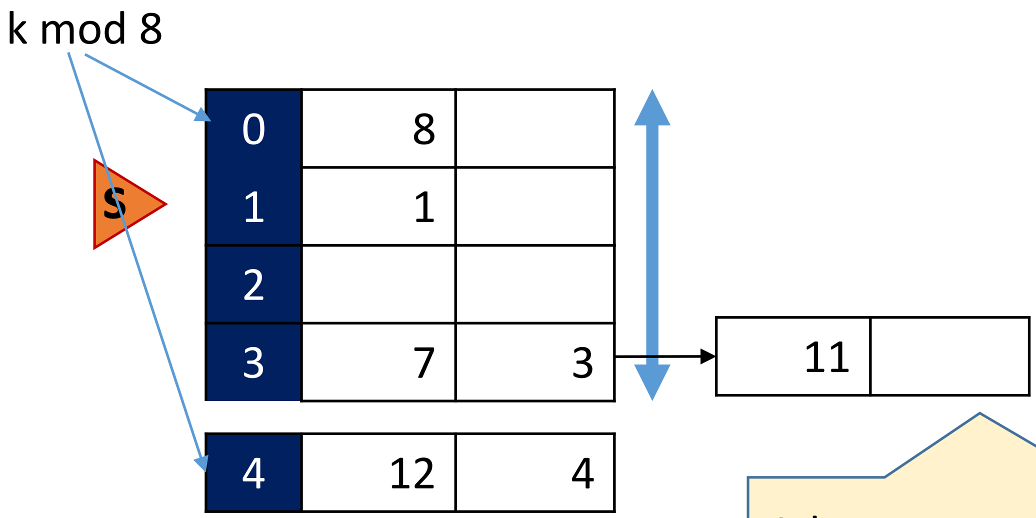


Use the next hash function h_2 = k mod 8 to redistribute the content of the current bucket between buckets 0 and 4: 4 and 12 are hashed to a new bucket. The overflow page is deleted.

Linear hashing: insert 11

The sequence of keys to be inserted





M = 5, hash capacity = 10

N = 7

i=1

Split when N/hash capacity > 3/4

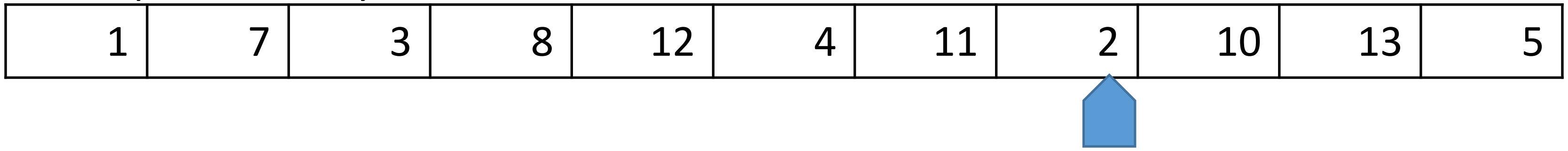
To find where a new record belongs, use h_1 (k) = k mod 4

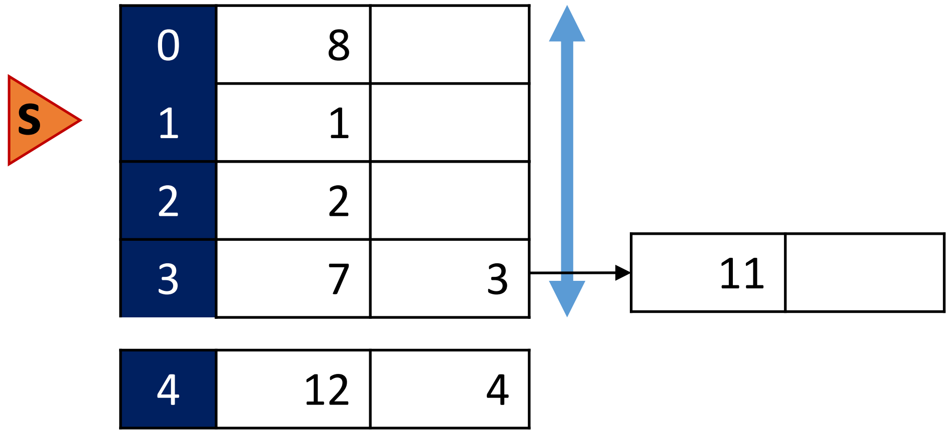
Advance s.
Now if a new key hashes to 0 – above s – then we find its place using mod 8, otherwise we placing it according to h_1(k) = k mod 4 – the hash function for the current level i=1

Linear hashing: insert 2

The sequence of keys to be inserted

1	7	3	8	12	4	11	2	10	13	5





M = 5, hash capacity = 10

N = 8

i=1

Split when N/hash capacity > 3/4

To find where a new record belongs, use h_1 (k) = k mod 4

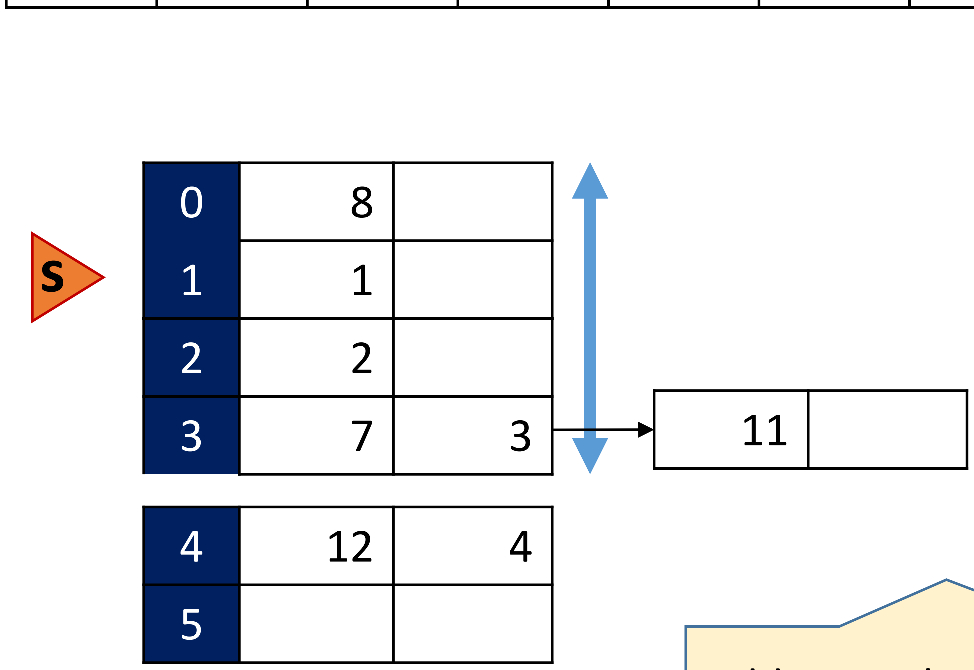
If h_1 < 1, use h_2 (k) = k mod 8

0.8 > 0.75. We need to split bucket 1.

Linear hashing: insert 2

The sequence of keys to be inserted

1	7	3	8	12	4	11	2	10	13	5



M = 6, hash capacity = 12

N = 8

i=1

Split when N/hash capacity > 3/4

To find where a new record belongs, use h_1 (k) = k mod 4

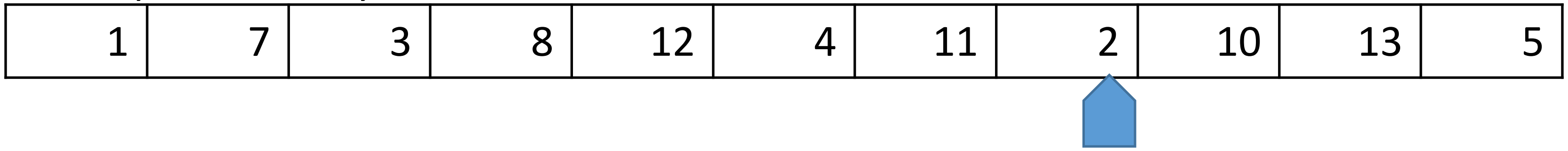
If $h_1 < 1$, use h_2 (k) = k mod 8

Add a new bucket. Redistribute content of bucket 1 between buckets 1 and 5, using the next-level hash function h_2 = k mod 8. Nothing to re-distribute

Linear hashing: insert 2

The sequence of keys to be inserted

1	7	3	8	12	4	11	2	10	13	5



M = 6, hash capacity = 12

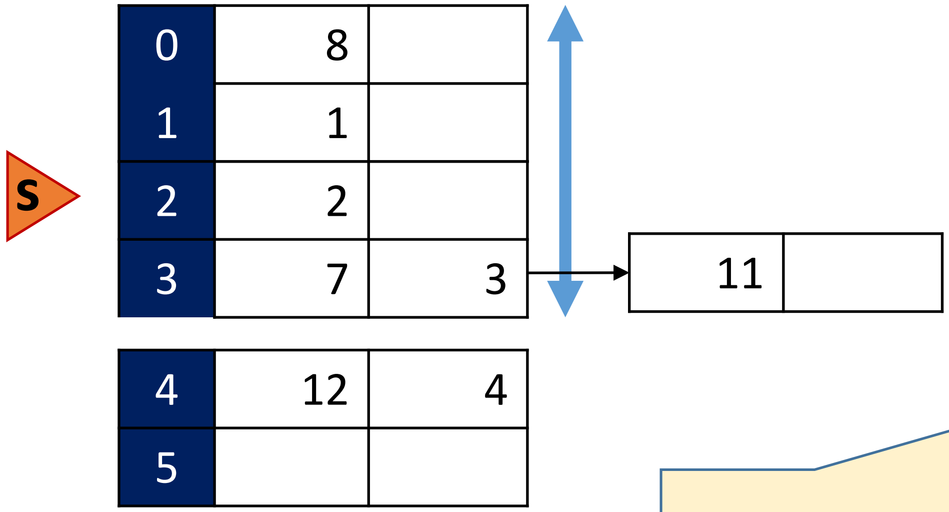
N = 8

i=1

Split when N/hash capacity > 3/4

To find where a new record belongs, use h_1 (k) = k mod 4

If h_1 < 1, use h_2 (k) = k mod 8

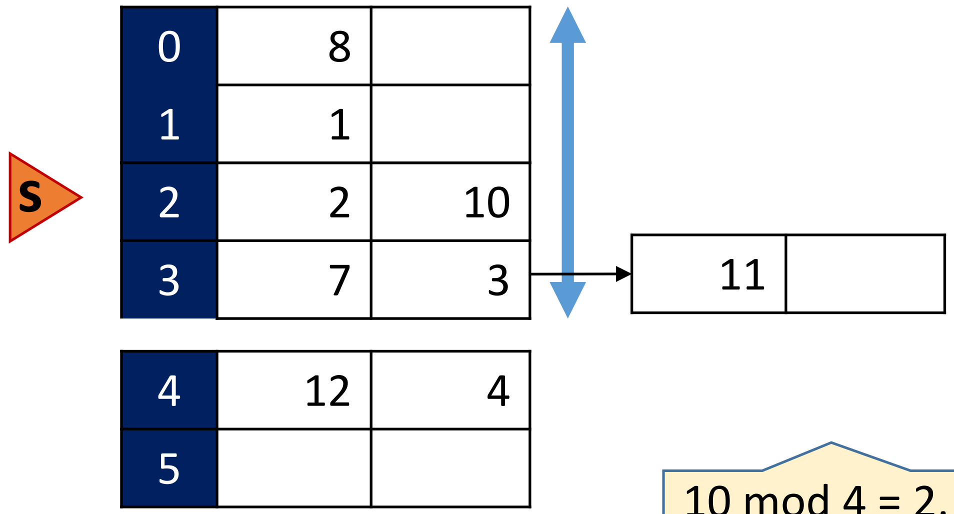


Advance s. Now if a key hashes to 0 or 1, use the next level hash function mod 8, otherwise put according to mod 4.

Linear hashing: insert 10

The sequence of keys to be inserted

1	7	3	8	12	4	11	2	10	13	5



M = 6, hash capacity = 12

N = 9

i=1

Split when N/hash capacity > 3/4

To find where a new record belongs, use h_1 (k) = k mod 4

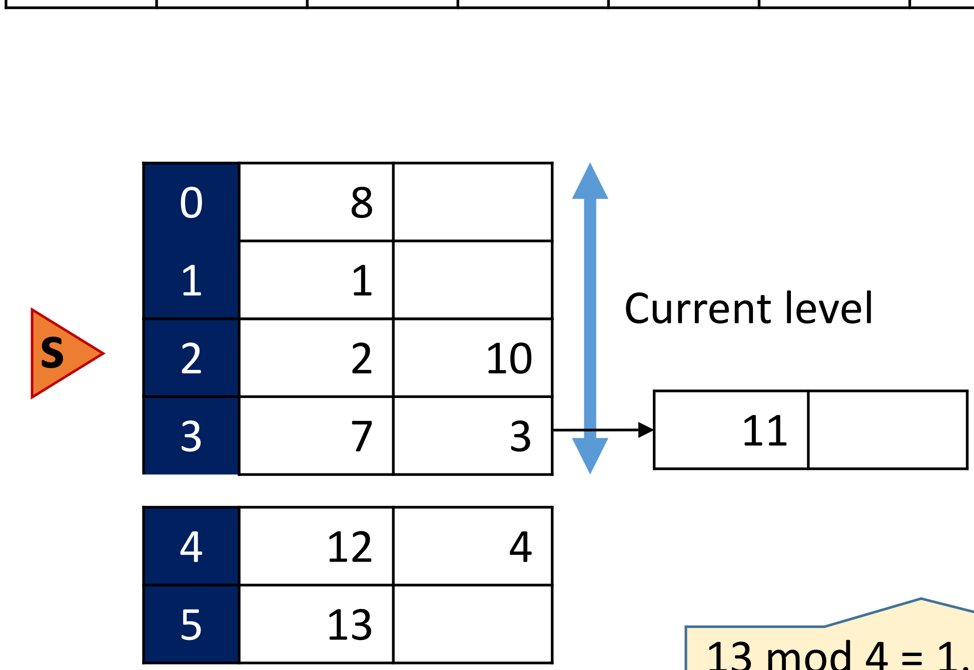
If h_1 < 1, use h_2 (k) = k mod 8

10 mod 4 = 2.
9/12 <= 3/4. No split needed

Linear hashing: insert 13

The sequence of keys to be inserted

1	7	3	8	12	4	11	2	10	13	5



M = 6, hash capacity = 12

N = 10

i=1

Split when N/hash capacity > 3/4

To find where a new record belongs, use h_1 (k) = k mod 4

If h_1 < 1, use h_2 (k) = k mod 8

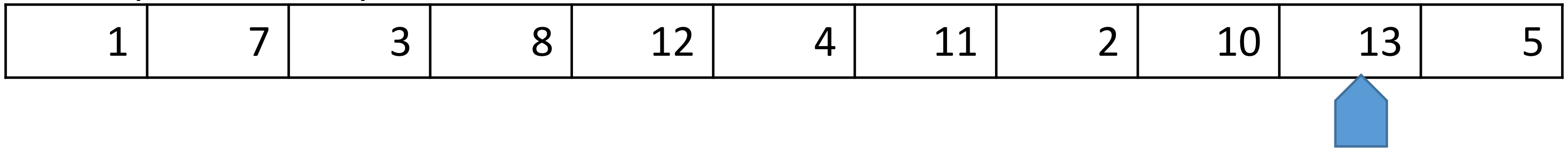
13 mod 4 = 1. Use 13 mod 8 = 5.
Check for split threshold: 10/12 > 3/4.
Need to split current bucket 2

etc. …

Try to finish this split and insert the next key.

Linear hashing: insert 13

The sequence of keys to be inserted



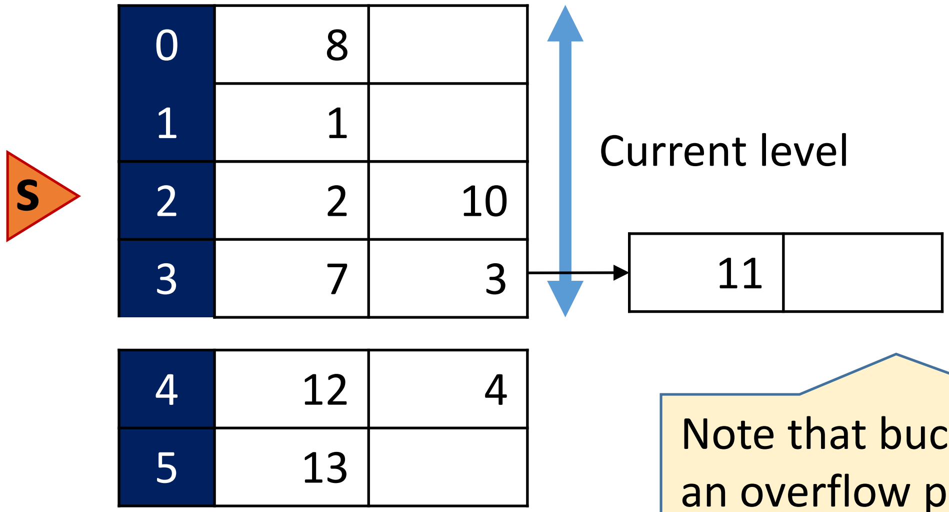
M = 6, hash capacity = 12

N = 10

Split when N/hash capacity > 3/4

To find where a new record belongs, use h_1 (k) = k mod 4

If h_1 < 1, use h_2 (k) = k mod 8

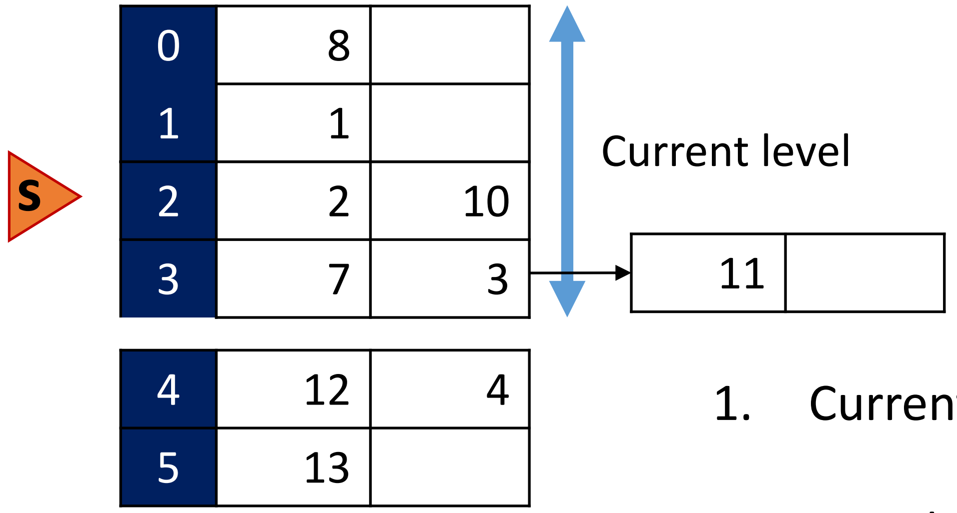


Note that bucket 3 still contains a link to an overflow page, however it will eventually be split in this round, and the keys re-distributed among new buckets

Linear hashing: notes

- Full buckets are not necessarily split
- Buckets that are split are not necessarily full
- Every bucket will be split sooner or later in the current round, and so overflows will be reclaimed and rehashed.
- Split pointer ***s*** decides which bucket to split next
  - At level *i*, ***s*** is between 0 and 2^i
  - ***s*** is incremented and after reaching 2^i is reset to 0. At this point all the buckets at level *i* have been split, and ***s*** will start a new round from 0 to 2^{i+1}
- Family of hash functions for each level: **$h_i(k) = h(k) \bmod (2^i * n)$**, where n is the initial number of buckets
- When level *i*+1 is reached, the capacity of the hash table at level *i* is doubled

Linear hashing: lookup example 1



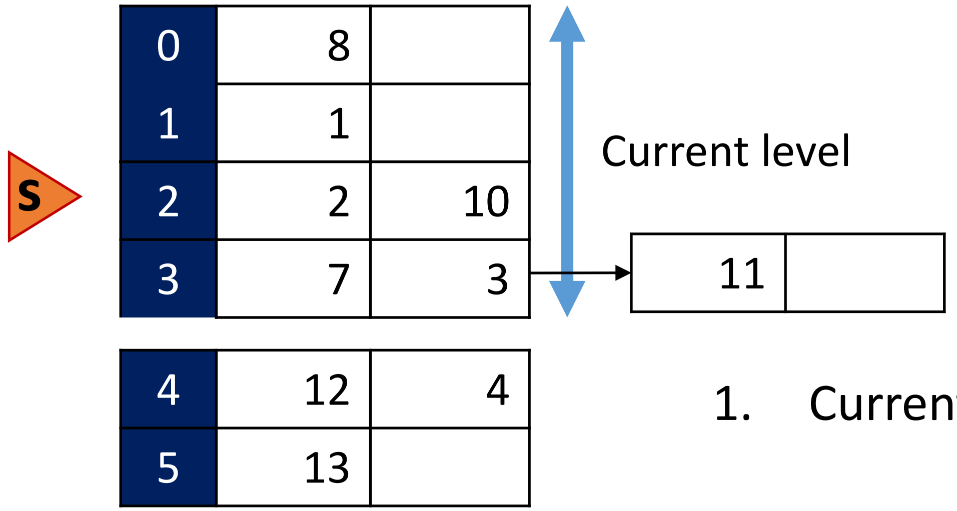
To find a record with key k:

Calculate h_1 (k) = k mod 4

If $h_1 < 2$, use h_2 (k) = k mod 8

1. Current level is 1. s= 2.
2. Search for key 5.
3. Compute $h_1(5)$ = 5 mod 4 = 1
   1 < s, compute $h_2(5)$ = 5 mod 8 = 5.
   Record with key 5 can be only in bucket 5.
4. Search inside bucket 5 – no such record

Linear hashing: lookup example 2



To find a record with key k:

Calculate h_1 (k) = k mod 4

If $h_1 < 2$, use h_2 (k) = k mod 8

1. Current level is 1. s= 2.
2. Search for key 7.
3. Compute $h_1(7)$ = 7 mod 4 = 3
   3 > s
   Record with key 7 can be only in bucket 3.
4. Search inside bucket 3 – yes!

Exercise (at home)

- Suppose we want to insert keys with hash values: 0000…1111 (0-15) in a linear hash table with 100% split threshold.
- Assume that a block can hold three records.

Summary

- Hash-based indexes: best for equality searches, do not support range searches.
- Static Hashing can lead to long overflow chains.
- ***Extendible Hashing*** avoids overflow pages by splitting a full bucket when a new data entry is to be added to it.
  - Directory to keep track of buckets, doubles periodically.
  - Can get large with skewed data*; additional I/O if this does not fit in main memory.
- ***Linear Hashing*** avoids directory by splitting buckets round-robin, and using overflow pages.

*For hash-based indexes, a *skewed* data distribution is one in which the *hash values* of data entries are not uniformly distributed!