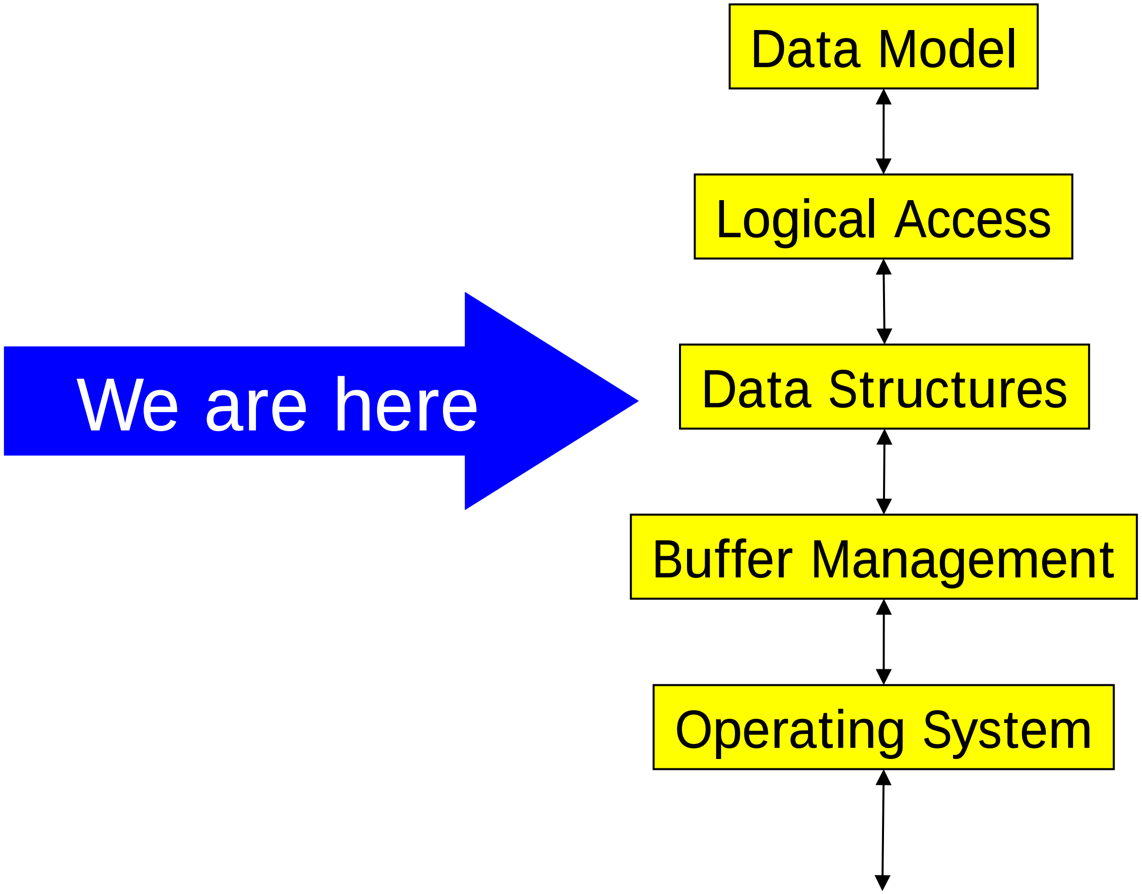
Datenbanksysteme II: Hashing

Ulf Leser

5 Layer Architecture



We are here

Data Model

Logical Access

Data Structures

Buffer Management

Operating System

Content of this Lecture

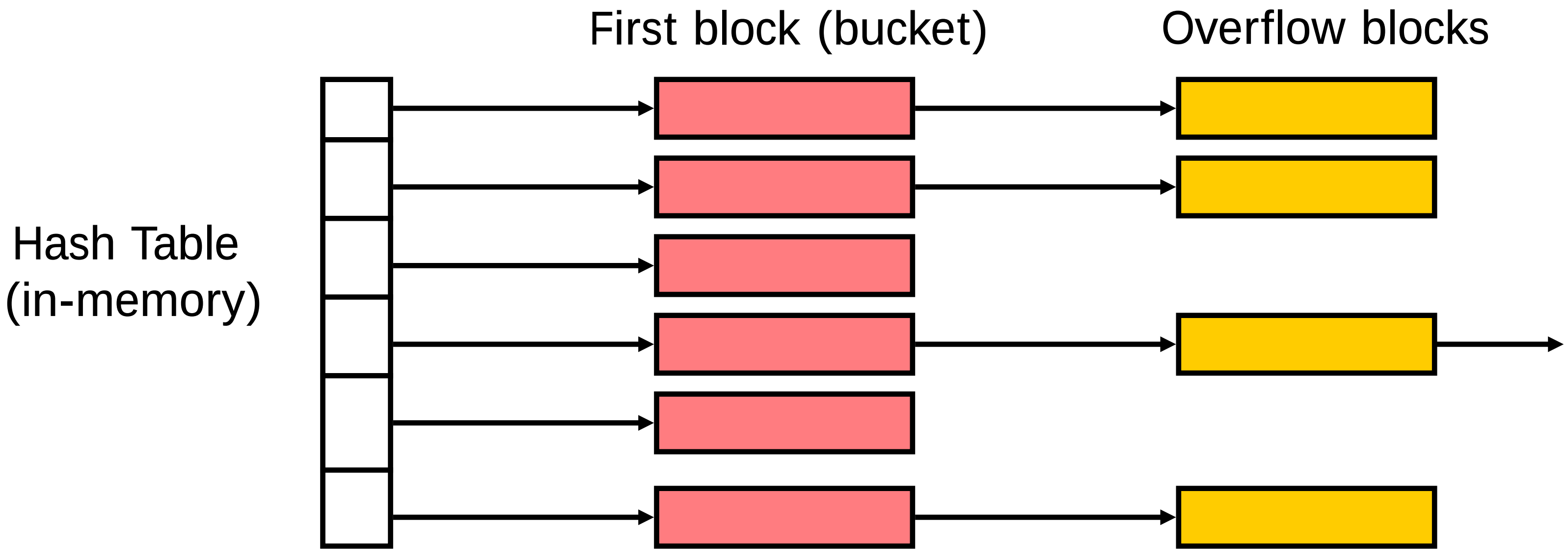
- Hashing
- Extensible Hashing
- Linear Hashing

Sorting or Hashing

- Sorted or indexed files
  - Typically log(n) IO for searching / deletions
  - Overhead for keeping order in file or in index
  - Maintaining low overhead (overflows) brings danger of degradation
  - Multiple orders require multiple indexes – multiple overhead
  - Good support for range queries
- Can we do better ... on average? ... under certain circumstances?
- Hash files
  - Can provide access in 1 IO
  - Can support searching for multiple attributes (with some overhead)
  - Incurs notable overhead if table size changes considerably
  - Are bad at range queries

Hash Files

- Set of buckets (≥ 1 blocks) B_0, …, B_{m-1} , m>1
- Hash function h(K) = {0 ,…, m-1} on a set K of values
- Hash table H (bucket directory) of size m with ptrs to B_i's
- Hash files are structured according to one attribute only

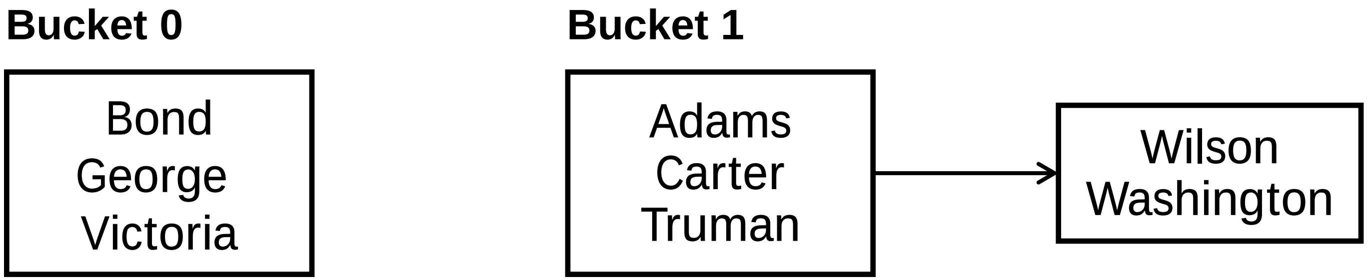


Example

- Hash function on Name

$$h\,(Name) = \begin{cases} 0 & \text{if last character} \leq M \\ 1 & \text{if last character} \geq N \end{cases}$$

Why last char?



Bucket 0

Bond
George
Victoria

Bucket 1

Adams
Carter
Truman

Wilson
Washington

Search “Adams”

1. h(Adams)=1
2. Bucket 1, Block 0?

Success

Search “Wilson”

1. h(Wilson)=1
2. Bucket 1, Block 0?
3. Bucket 1, Block 1?

Success

Search “Elisabeth”

1. h(Elisabeth)=0
2. Bucket 0, Block 0?

Failure

Efficiency of Hashing

- Given: n records, R records per block, m buckets
- Assume hash table is in main memory
- Average number of blocks per bucket: n / (m*R)
  - Assuming a (perfect) uniformly distributing hash function
- Search
  - n / (m*R) / 2 for successful search
  - n / (m*R) for unsuccessful search
- Insert
  - n / (m*R) if end of bucket cannot be accessed directly
  - n / (2m*R) if free space in one of the bucket
- If |H|=m large enough and good hash function: 1 IO

Hash Functions

- Examples: Modulo, Bit-Shifting
- Desirable: Uniform mapping of hash keys onto m
- "Ideal" (i.e. uniform) mapping possible if data distribution and number of records are known in advance
  - Which is unusual – data changes
- Application-dependent hash functions
  - Incorporating knowledge on expected distribution of keys

Problems with Hashing

- Hashing may degenerate to sequential scan
  - If number of buckets static and too small
  - If hash function produces large skew
- Extending hash range requires complete rehashing
- No efficient range queries
  - Requires enumerating all distinct values in range
- Very powerful, if everything works fine
- “Almost constant” access time

Content of this Lecture

- Hashing
- Extensible Hashing
- Linear Hashing

Extensible Hashing

- Traditionally, hashing is a static index structure
  - Structure (buckets, hash function) is fixed once and never changed
- To be used in DBS, hash tables/function must adapt to changing data volumes and value distributions
- Principle idea of Extensible Hashing
  - Hash function generates (long) bitstring
    - Should distribute values evenly on every position of bitstring
  - Only a prefix of this bitstring as index in hash table
  - Size of prefix adapts to number of records
    - As does size of hash table
  - Different buckets use different prefix sizes

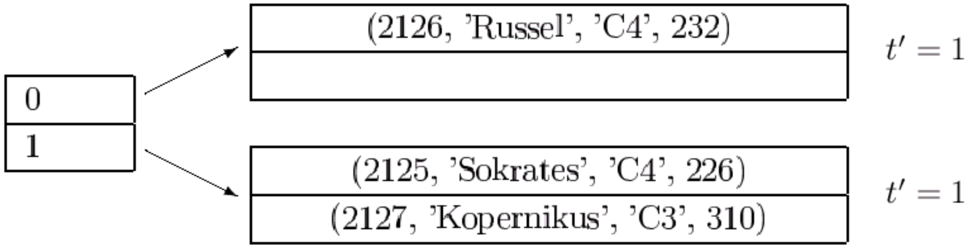
Hash functions

- h: K → {0,1}*
- Size of bitstring should be long enough for mapping into as many buckets as maximally desired
  - Though we do not use them all most of the time
- Example: inverse person IDs
  - h(004) = 001000000... (4=0..0100)
  - h(006) = 011000000... (6=0..0110)
  - h(007) = 111000000... (7 =0..0111)
  - h(013) = 101100000... (13 =0..01101)
  - h(018) = 010010000... (18 =0..010010)
  - h(032) = 000001000... (32 =0..0100000)
  - H(048) = 000011000… (48 =0..0110000)

Extensible Hashing

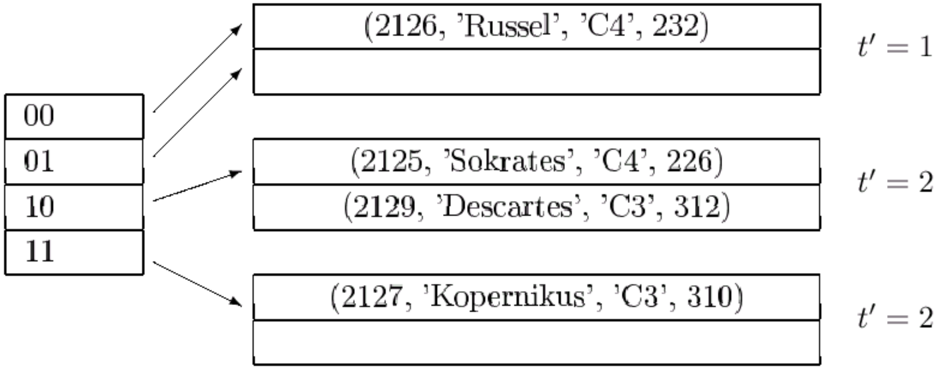
- Parameters
  - d: global „depth“ of hash table, size of longest prefix currently used
  - t: local „depth“ of each bucket, size of prefix used in this bucket
- Example
  - Let a bucket store two records
  - Start with two buckets and 1 bit for identification ($d=t_1=t_2=1$)

Keys	as bitstring	inverse	$h_{d=1}(k)$
2125	100001001101	101100100001	1
2126	100001001110	011100100001	0
2127	100001001111	111100100001	1



Example cont‘d

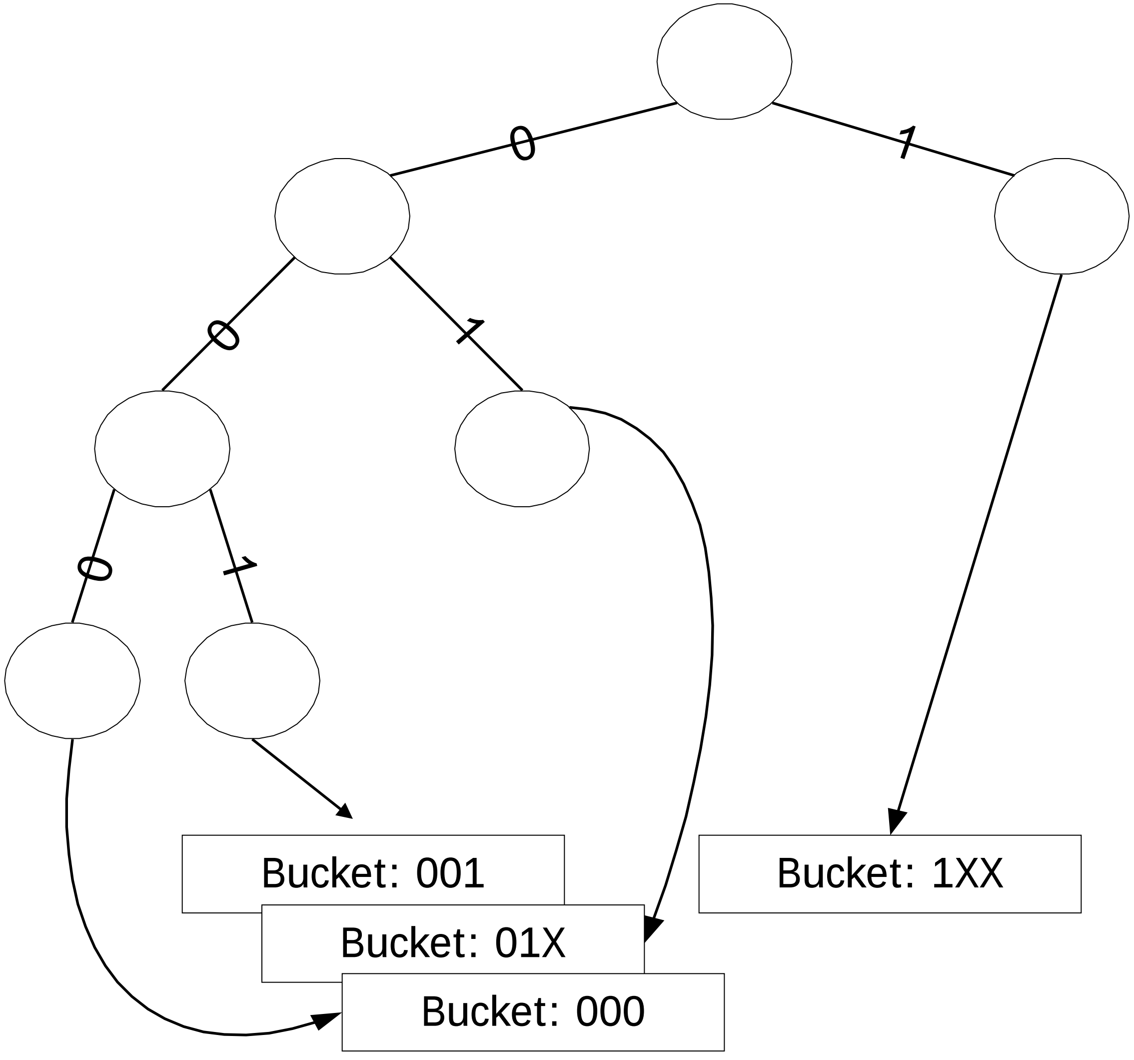
k	as bitstring	inverse	$h_{d=1}$
2125	100001001101	101100100001	1
2126	100001001110	011100100001	0
2127	100001001111	111100100001	1
2129	100001010001	100010100001	1



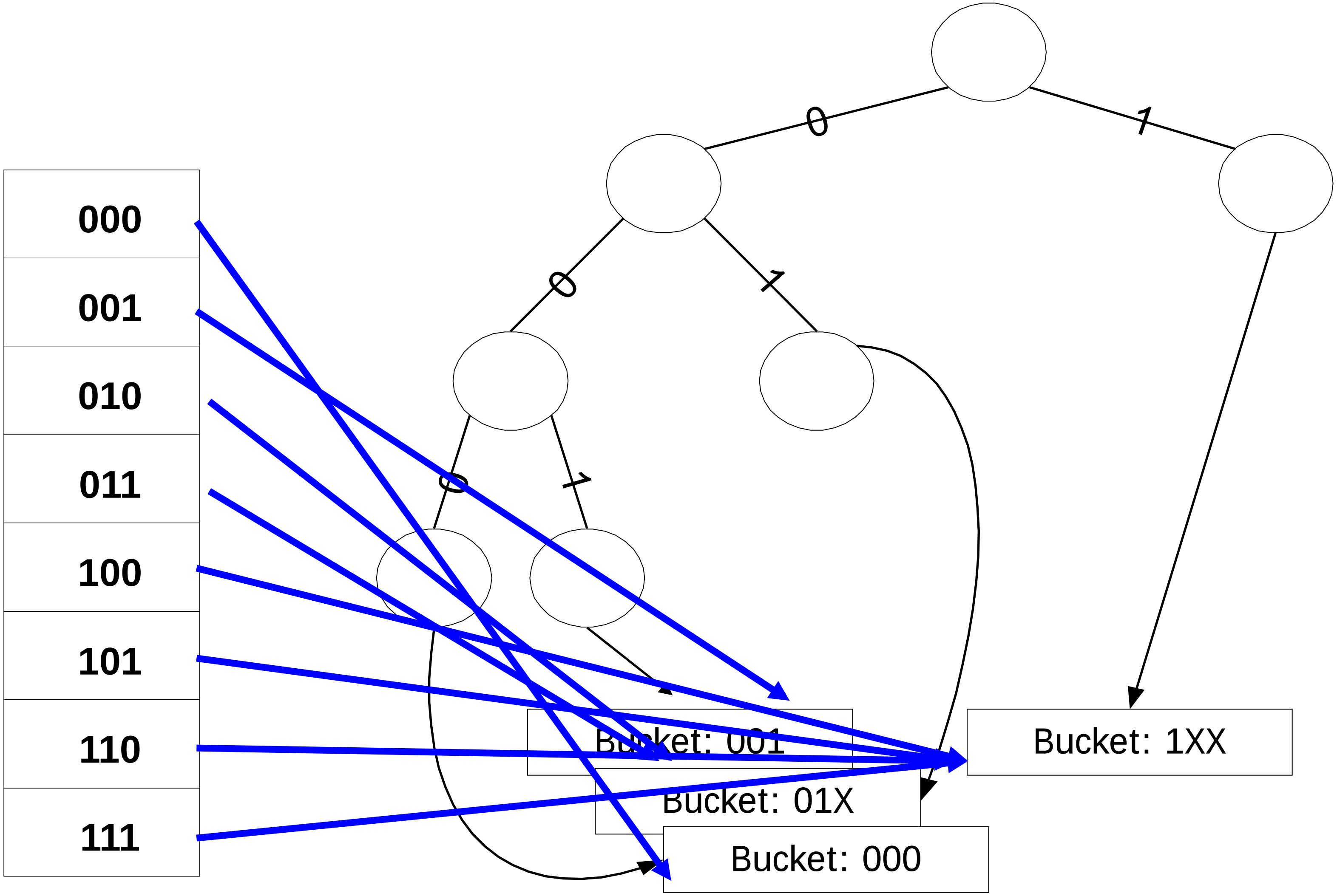
- New record with x=2129
- Bucket for „1“ full
- Need to split
  - Duplicate hash table, d++
  - Pointers to un-splitted blocks remain unchanged
  - Overflowing bucket is split and records are distributed according to bits until new d

More Complex Example

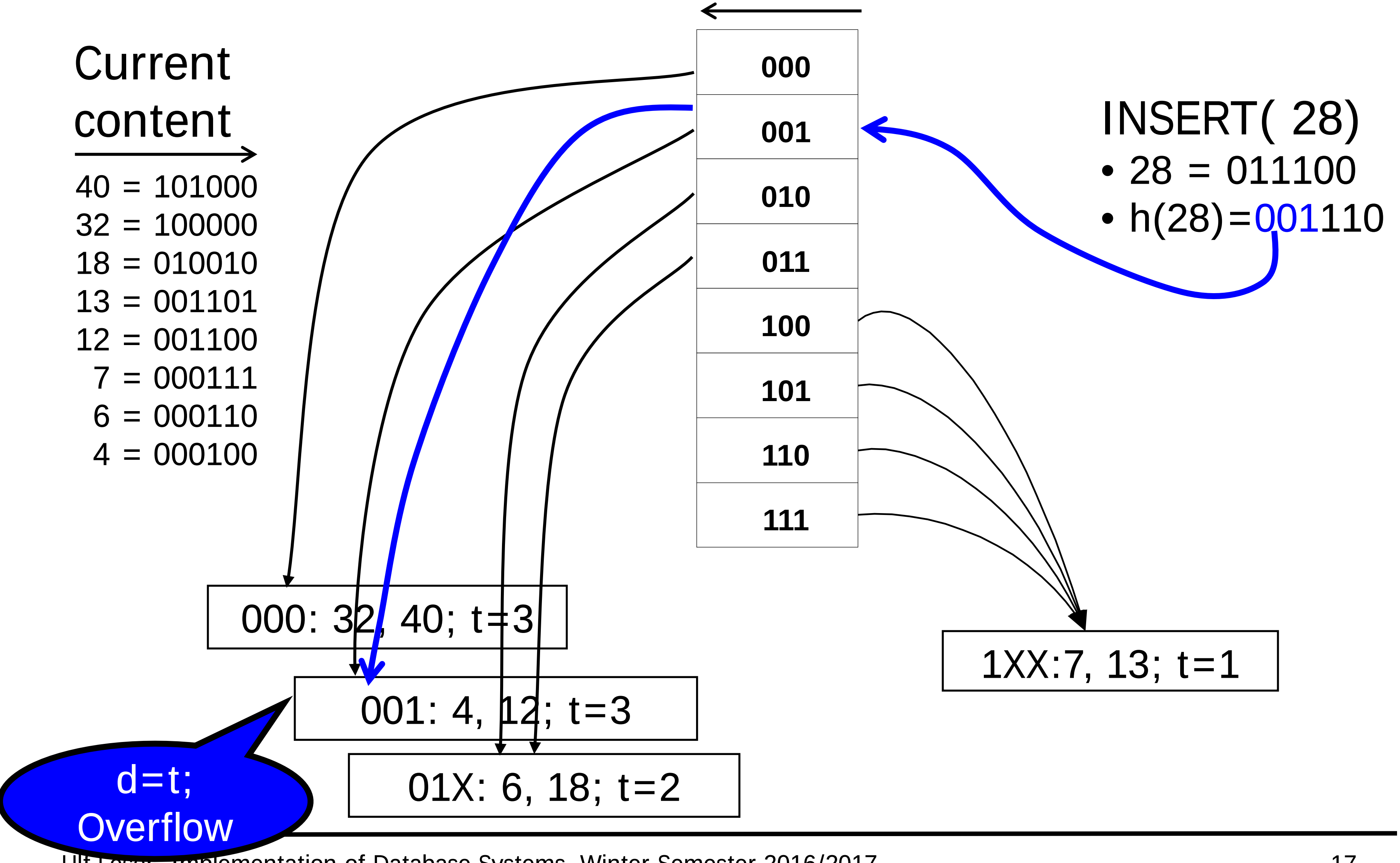
- Assume reversed bit hash function on integers
- Currently four buckets in use
- Global depth d=3
- Local depth t between 1 and 3
- Size of global directory: $2^d=8$



Example: Hash Table



Inserting Values



Current content

40 = 101000
32 = 100000
18 = 010010
13 = 001101
12 = 001100
7 = 000111
6 = 000110
4 = 000100

INSERT(28)

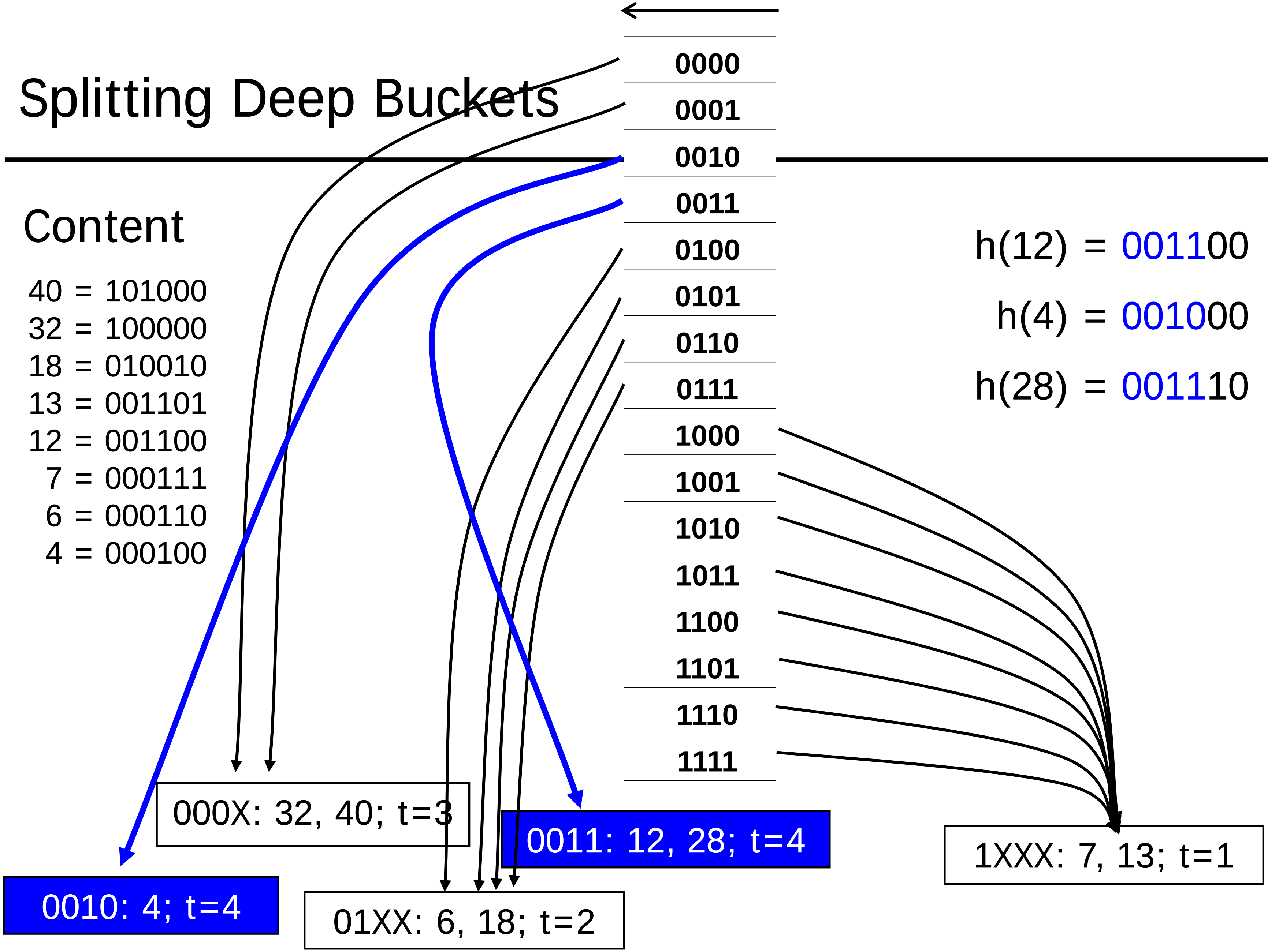
- 28 = 011100
- h(28)=001110

Splitting Deep Buckets

Content

40 = 101000
32 = 100000
18 = 010010
13 = 001101
12 = 001100
7 = 000111
6 = 000110
4 = 000100

h(12) = 001100
h(4) = 001000
h(28) = 001110



0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

000X: 32, 40; t=3

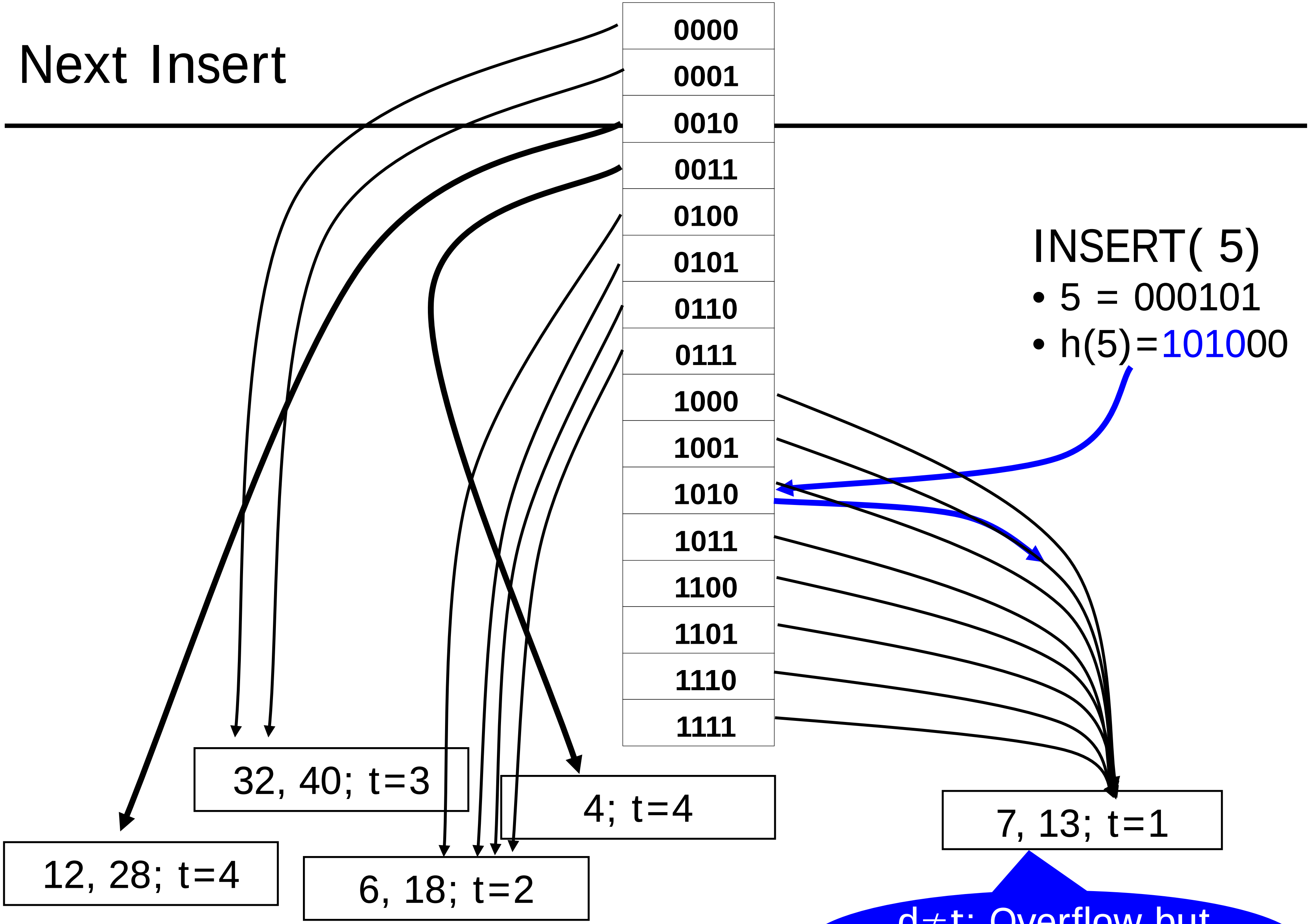
0011: 12, 28; t=4

1XXX: 7, 13; t=1

0010: 4; t=4

01XX: 6, 18; t=2

Next Insert



0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

32, 40; t=3

12, 28; t=4

4; t=4

6, 18; t=2

7, 13; t=1

INSERT(5)

- 5 = 000101
- h(5)=101000

d≠t: Overflow but no dir duplication

Splitting Shallow Buckets

- Assume we have to split overflowing bucket B
- B is shallow: t<d
- For all records r∈B, h(r) has the same length-t prefix
- If we split at next position (t++)
  - Generate new bucket and rehash records
  - This might generate an empty bucket
    - May be suppressed: NULL in hash table
  - The other bucket might still be overflowing – repeat split
    - In the example, we rehash 5=101000, 7=111000, 13=101100
    - Hence, one split suffices (with block prefixes 10 and 11)
    - But, if we had 5=10100, 13=101100, 21=101010?
- Might eventually force a deep split with increase in d
- Suboptimal space usage (many almost empty buckets)

Summary

- Advantages
  - Adapts to growing or shrinking number of records
    - Deletion not shown – think yourself
  - No rehashing of the entire table – only overflown bucket
  - Very fast if directory can be cached and h is well chosen
- Disadvantages
  - Directory needs to be maintained (locks during splits, storage ...)
  - Does not properly handle skew wrt hash function
    - No guaranteed bucket fill degree
      - Many buckets might be almost empty, few almost full
    - Directory can grow exponentially for linearly more records
      - If all records share a very long prefix
  - Values are not sorted, no range queries
- Use for uniformly distributed data with proper hash function

Content of this Lecture

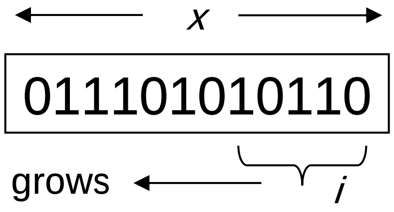
- Hashing
- Extensible Hashing
- Linear Hashing

Linear Hashing

- Similar scheme as in extensible hashing, but
  - Don’t double directory on overflow, but increase one-by-one
  - Guaranteed lower bound on bucket fill-degree
  - Tolerate some overflow blocks in buckets
    - Few on average if hash function spreads evenly

Overview

- h generates bitstring of length x, read right to left
- Parameters
  - i: Current number of bits from x used
    - As i grows, more bits are considered
    - If h generates x bits, we use $a_1a_2...a_i$ for the last i bits of h(k)
  - n: Total number of buckets currently used
    - Only the first n values of bitstrings of length i have their own buckets
  - r: Total number of records



- Fix threshold t – linear hashing guarantees that r/n<t
  - As r increases, we sometimes increase n such that always r/n<t
  - Linear hashing only guarantees the average fill-degree
    - But does not prevent chaining in case of “bad” hash function
  - Restricts the average #buckets that must be searched (not WC)

Insert(k): First Action

- Insert new record with key k
  - Let m by the integer value encoded by the i last bits of h(k)
  - If m<n
    - Hence, the target bucket exists
    - Store k in bucket m, potentially using overflow blocks
  - If m≥n
    - Bucket m does not exist
      - There exist buckets 0 ... n-1
    - We redirect k into a bucket that does exist
    - Flip i-th bit (from the right) of m to 0 and store k in this bucket
      - Algorithm ensures that here the i'th bit must be 1
    - This flipping also needs to be done when searching keys

Insert(k): Second Action

- Check threshold; if $r/n \geq t$, then
  - If $n=2^i$
    - No more room to add another bucket
    - Set i++
    - This is only a conceptual increase – no physical action
    - Proceed (now we have $n<2^i$)
  - If $n<2^i$
    - There is still (now) room on our address space
    - We add (n+1)th bucket and set n++
    - We need to choose which bucket to split
      - We do not split the bucket where we just inserted (why should we?)
      - We do not search for overflowed buckets (too costly)
      - Instead, we use a cyclic scheme

Which Bucket to Split

- We split buckets in fixed, cyclic order
- Split bucket with number $n-2^{i-1}$
  - As n increases, this pointer cycles through all buckets
  - Let $n=1a_2a_3...a_i$; then we split block with ID $a_2a_3...a_i$ into two blocks with ID $0a_2a_3...a_i$ and ID $1a_2a_3...a_i$
    - Requires redistribution of bucket with hash key $a_2a_3...a_i$
    - This is one of the buckets where we had put redirected records
    - This is not necessarily an overflowed bucket
    - Recall: Only the average fill degree is guaranteed

Buckets Split Order

Assume we would split after every insert

i	n	Existing buckets	Bucket to split: $n-2^{i-1}$	Generates
1	2=10	0,1	0	00 10
2	3=11	00,10 1	1	01 11
	4=100	00,10 01,11	00	000 100
3	5=101	000,100 10, 01,11	01	001 101
	6=110	000,100 001,101 10,11	10	010 110
	7=111	000,100,001,101, 010,110, 11	11	011 111

Example

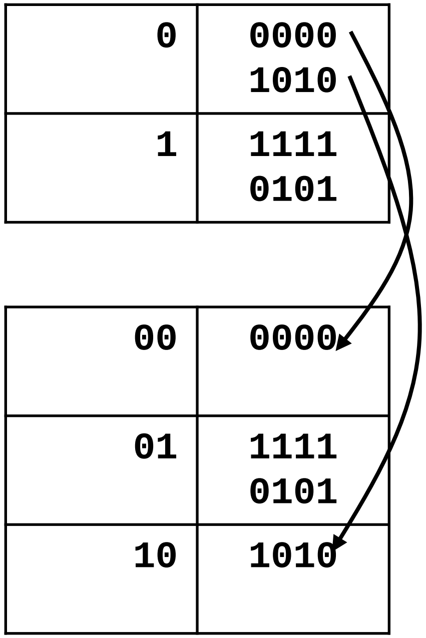
- Assume 2 records in one block, x=4, t=1.74, i=1

Start (with arbitray keys)

0	0000 1010
1	1111

1a) Insert 0101
$m=1<n=10_b$
Insert into bucket 1
But now $r/n \geq t$

1b) Since $n=2^i=2=10_b$
We need more address space
Increase i (virtually)
Add bucket number $2=10_b$
$n=10_b=1a_1$: Split bucket 0
into 10 and 00
n++



0	0000 1010
1	1111 0101

00	0000
01	1111 0101
10	1010

01: Yet unsplit
stores 01 and 11
(by flipping)

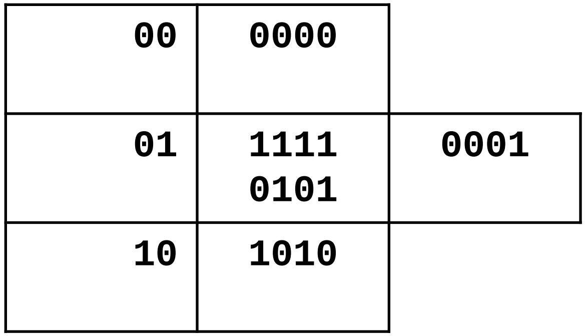
Example 2

2) Insert 0001

m=1, bucket exists

Insert into m

Requires overflow block



3a) Insert 0111

$m=3=n=11_b$

Bucket doesn’t exist

Flip and redirect to 01

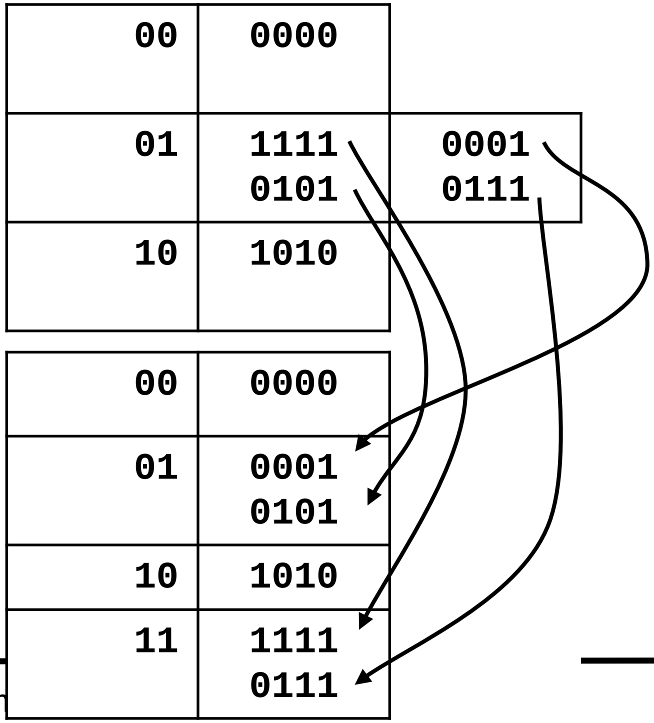
3b) $r/n=6/3 \geq t$ – We split

n<4, so no need to increase i

Add bucket number 3=11b

Since $n=3=11_b$, with split 01

Delete overflow block



Example 3

4a) Insert 0011
$m=3=11_b < n=4=100_b$
Insert into 11_b

00	0000	
01	0001 0101	
10	1010	
11	1111 0111	0011

4b) We must split again
Since $n=2^i$, increase i
Nothing to do physically
(“Think” a leading 0)

00	0000	
01	0001 0101	
10	1010	
11	1111 0111	0011

Example 4

4c) Split

- Add block number $4=100_b$
- Split 000_b into 000_b and 100_b

000	0000	
001	0001 0101	
010	1010	
011	1111 0111	0011
100	-	

We keep the average bucket filling
But we have unevenly filled buckets – some empty, some overflow

Observations

- Due to the extension mechanism: $2^{i-1} \leq n \leq 2^i$
  - Whenever n reaches 2^i, i is increased => 2^i doubles and $n=2^i/2$ (for the new i)
  - Hence, n as binary number always has the form $1b_1b_2\ldots b_{i-1}$
- As defined: $m<2^i$
  - But possibly: $m>n$
    - Such m must have a leading 1, as n must have one (see prev observation)
    - If we drop the leading 1 in m, we get $m_{new}<m/2$
    - Since $n \geq 2^{i-1}$, $m_{new} \leq n$
    - Thus, the chosen bucket m_{new} must already exist

Summary

- Advantages
  - Adapts to varying number of records
  - Slower growth and on average better space usage compared to extensible hashing
  - If buckets are sequential on disk, we don't need a directory
    - Compute m: look in m'th bucket (possible after flipping)
- Disadvantages
  - Can degrade, as buckets are split in fixed order
  - No adaptation to skewed value distribution
  - Creates random IO on disk through overflow blocks