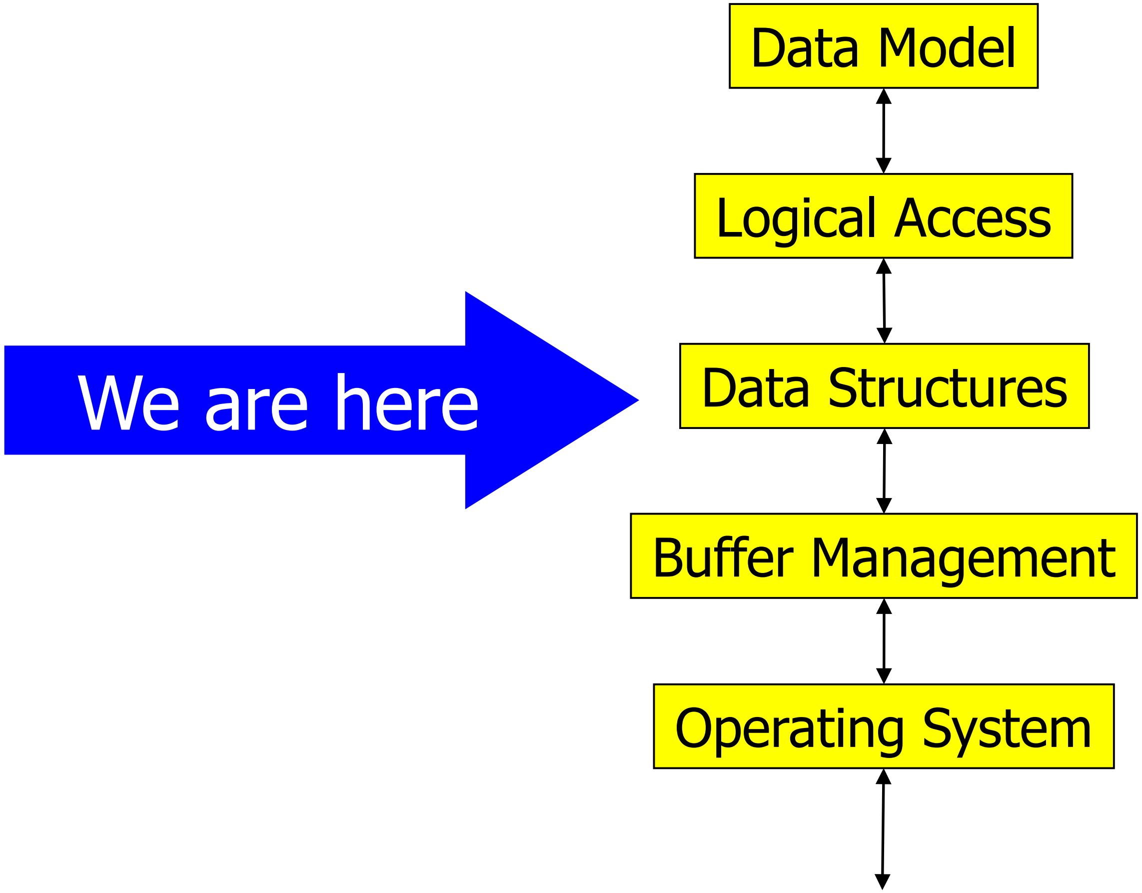
Datenbanksysteme II: Dynamic Hashing

Ulf Leser

5 Layer Architecture



We are here

Data Model

Logical Access

Data Structures

Buffer Management

Operating System

Content of this Lecture

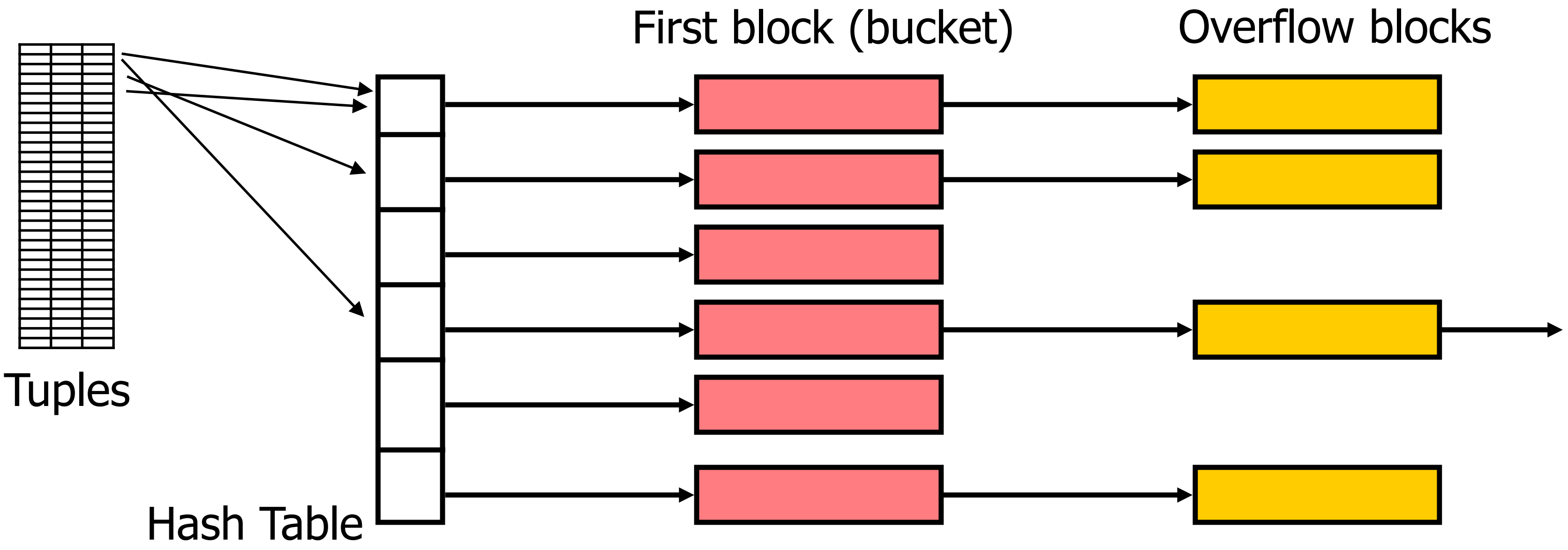
- Hashing
- Extensible Hashing
- Linear Hashing

Sorting or Hashing

- Sorted or indexed files
  - Typically log(n) IO for searching / deletions
  - Overhead for keeping order in file or in index
    - Danger of degradation
    - Multiple orders require multiple indexes – multiple overhead
  - Good support for range queries
- Can we do better … under certain circumstances?
- Hash files
  - Can provide key-based access in 1 IO
    - Searching for multiple keys – multiple hash indexes
  - Incurs notable overhead if table size changes considerably
    - Dynamic hashing
  - Are bad for range queries

Hash Files

- Set of buckets (≥ 1 blocks) $B_0, \ldots, B_{m-1}$, m>1
  - We hash keys to blocks, not to single tuples
  - We need to search key inside block / bucket
- 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

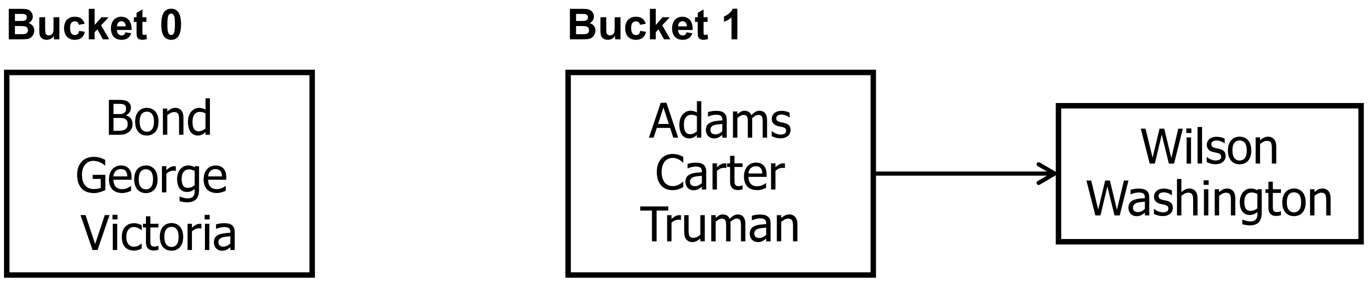


Example

- Hash function on Name

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

Why last char?



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

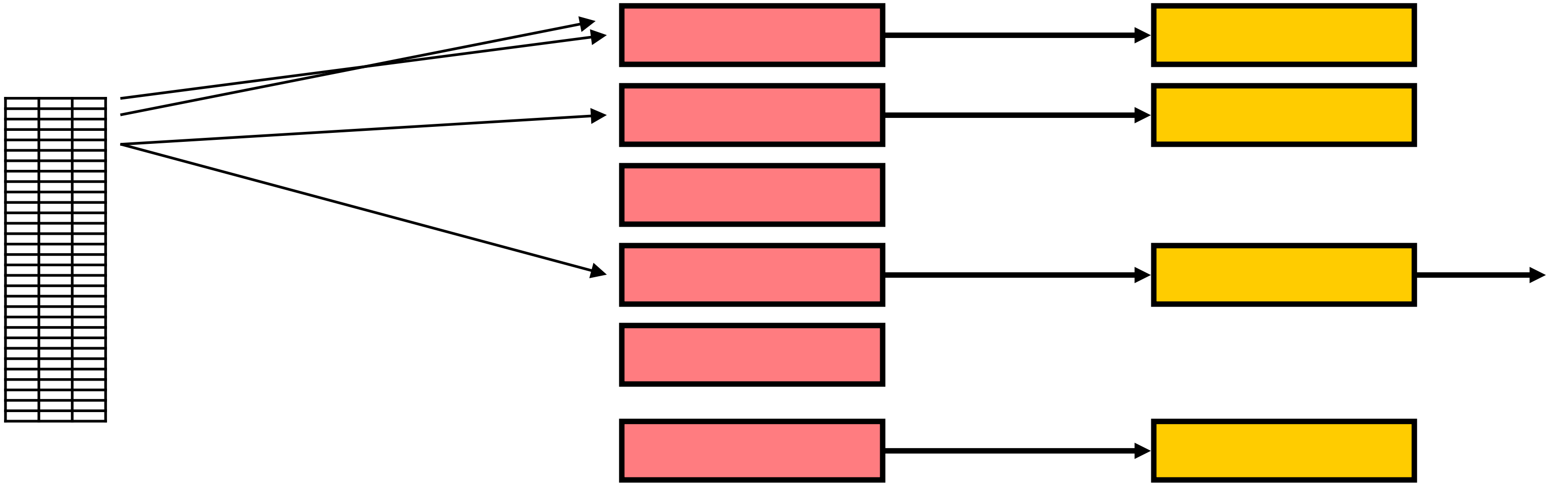
Search "Elisabeth"

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

Failure

Alternative: Direct Block Hashing

- We can also directly hash keys into (first) block number
  - Requires consecutive range of blocks
  - h(key) = BLOCK_OFFSET + h'(key)
- Removes need for main memory hash table
- Heavily restricts block placement on disk
  - Inappropriate for fast changing data



Efficiency of Hashing

- Given n records, R records per block, m buckets
- Assume H is in main memory (or there is none)
- Average number of blocks per bucket: n / (m*R)
  - Assuming a uniform hash function and no empty space
  - Difficult to achieve in practice
- Search IO complexity
  - n / (m*R) / 2 for successful search
  - n / (m*R) for unsuccessful search (entire bucket)
- Insert IO complexity
  - n / (m*R) if end of bucket cannot be accessed directly
  - n / (m*R) / 2 if free space in one of the bucket
- If m large enough and good hash function: 1 IO

Hash Functions

- Examples: Modulo, Bit-Shifting, aggregates, …
- Desirable: Uniform mapping of keys into [0…m-1]
  - Keys should be equally distributed over all blocks – all the time
- Uniform mapping only possible if data distribution and number of records (for estimating m) known in advance
  - Which is unusual
- If known: Application-dependent hash functions
  - Incorporating knowledge on expected distribution of keys

Properties

- Hashing may degenerate to sequential scan
  - If number of buckets static and too small
  - If hash function produces large bias
- Extending m requires complete rehashing
  - We need a new hash function
  - Blocks all operations on this table
- Inefficient range queries – scan
  - Or enumerate all distinct values in range (only integer)
- Very fast iff everything works fine
  - "Practically constant" IO complexity

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
  - Cannot be changed gracefully (with small & local overhead)
- For DBMS, hashing must adapt to changing data volumes and value distributions
  - Dynamic hashing
- First idea: Extensible Hashing
  - Hash function generates (long) bitstring
    - Should distribute values evenly on every position of bitstring
  - Only a prefix of this bitstring is used as index in hash table
  - Size of prefix adapts to number of records
    - As does size of hash table

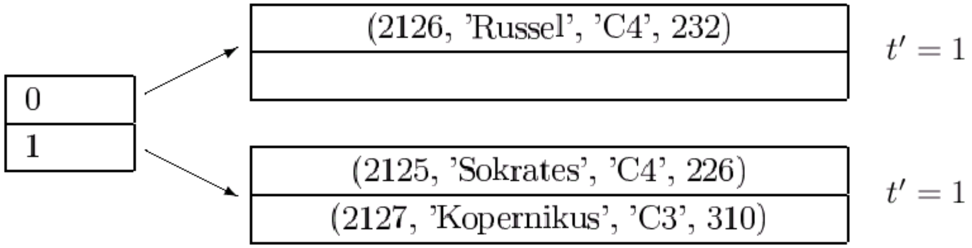
Hash functions

- 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

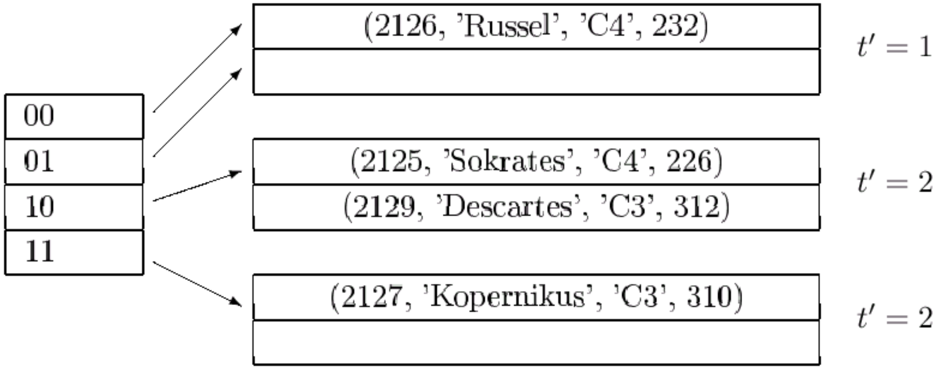
- Parameters
  - d: global „depth“ of hash table, size of longest prefix currently used
  - t: local „depth“ of a bucket, size of prefix used in this bucket
- Example
  - Let a bucket store two records
  - Start with two buckets and 1 bit for identification ($d=t_1=t_2=1$)

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



Example cont‘d

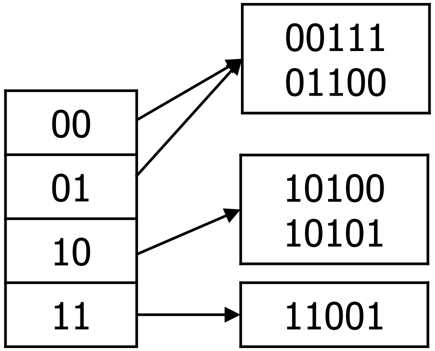
k	as bitstring	inverse	$h_{d=1}$
2125	100001001101	101100100001	1
2126	100001001110	011100100001	0
2127	100001001111	111100100001	1
2129	100001010001	100010100001	1

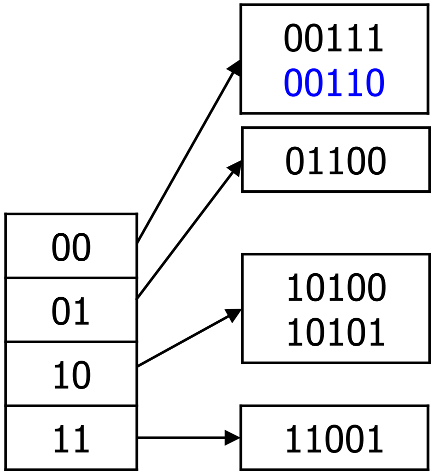


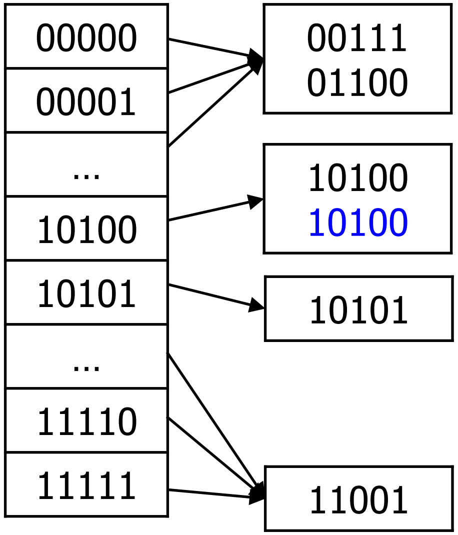
- New record with x=2129
- Bucket for „1“ is full
- Need to split
  - Duplicate hash table, d++
    - We conceptually have four buckets
  - Un-splitted blocks remain unchanged
  - Overflowing bucket is split and records are distributed according to next bit

Special Cases

- If block b overflows and t(b)<d
  - Create two new buckets, leave d unchanged
  - Distribute data from d according to bit t(d) and t(d)++
- If distribution creates one overflown and one empty bucket
  - Recurse – split overflown bucket again (and again and again …)

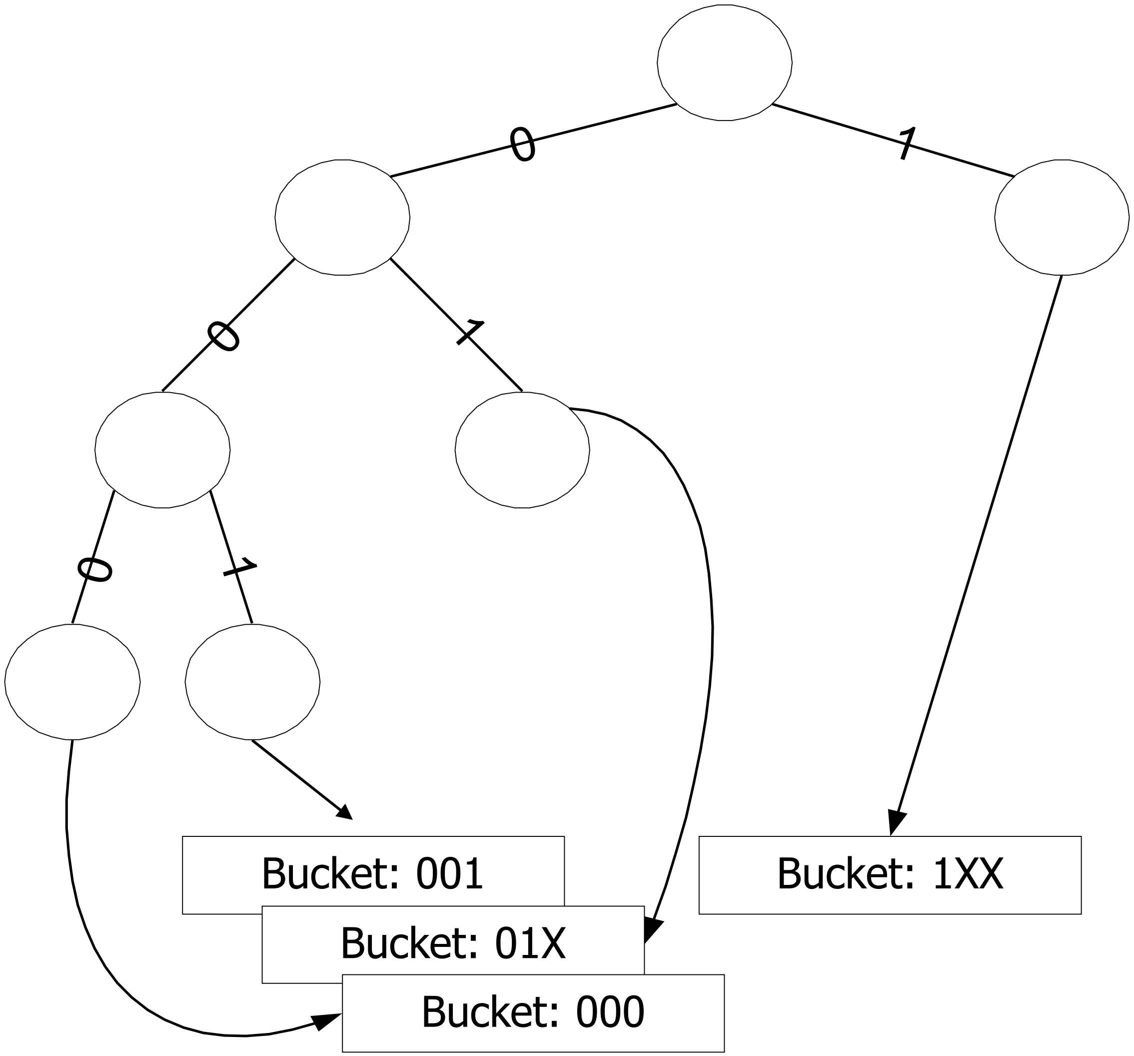




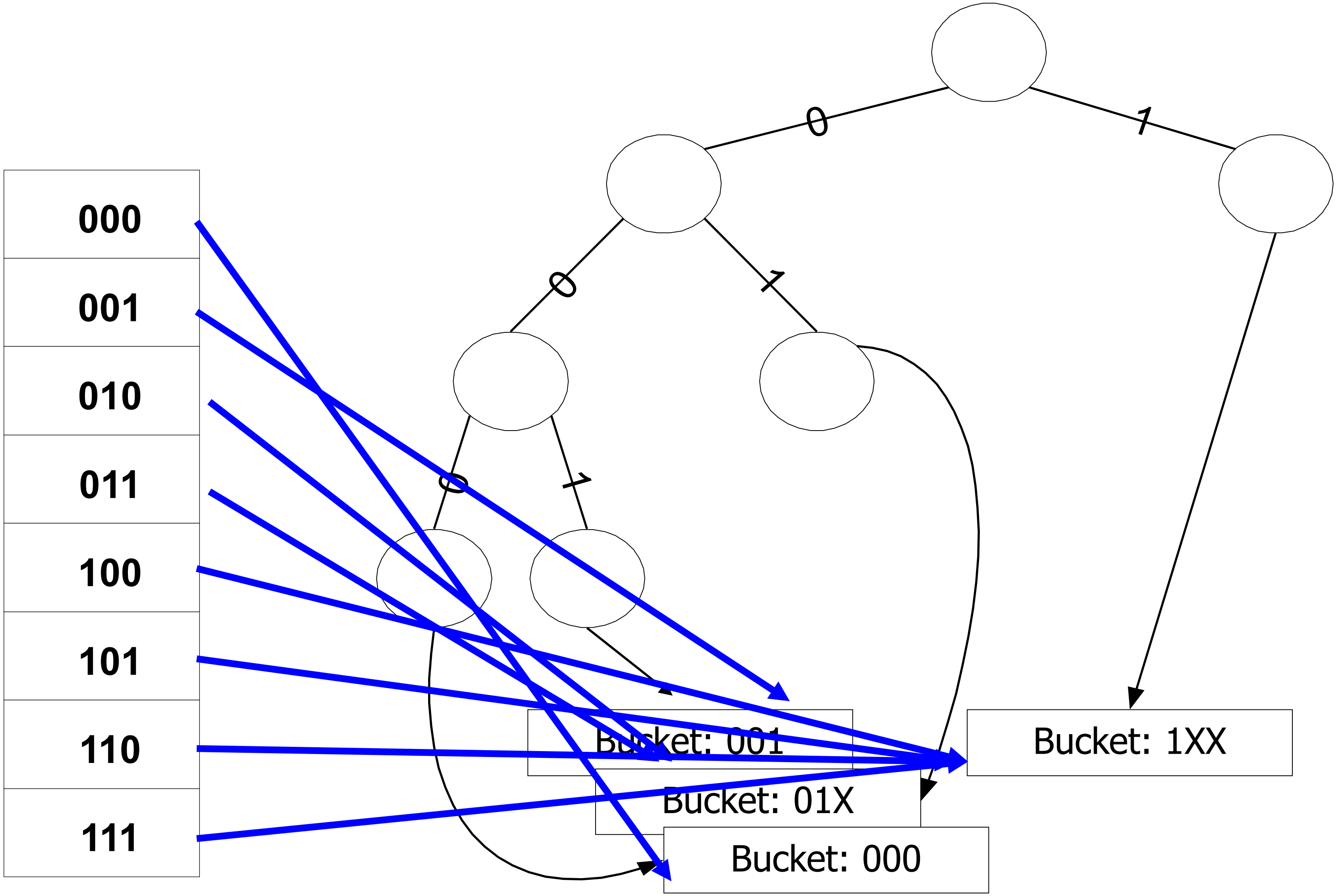


More Complex Example

- 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: $2^d=8$



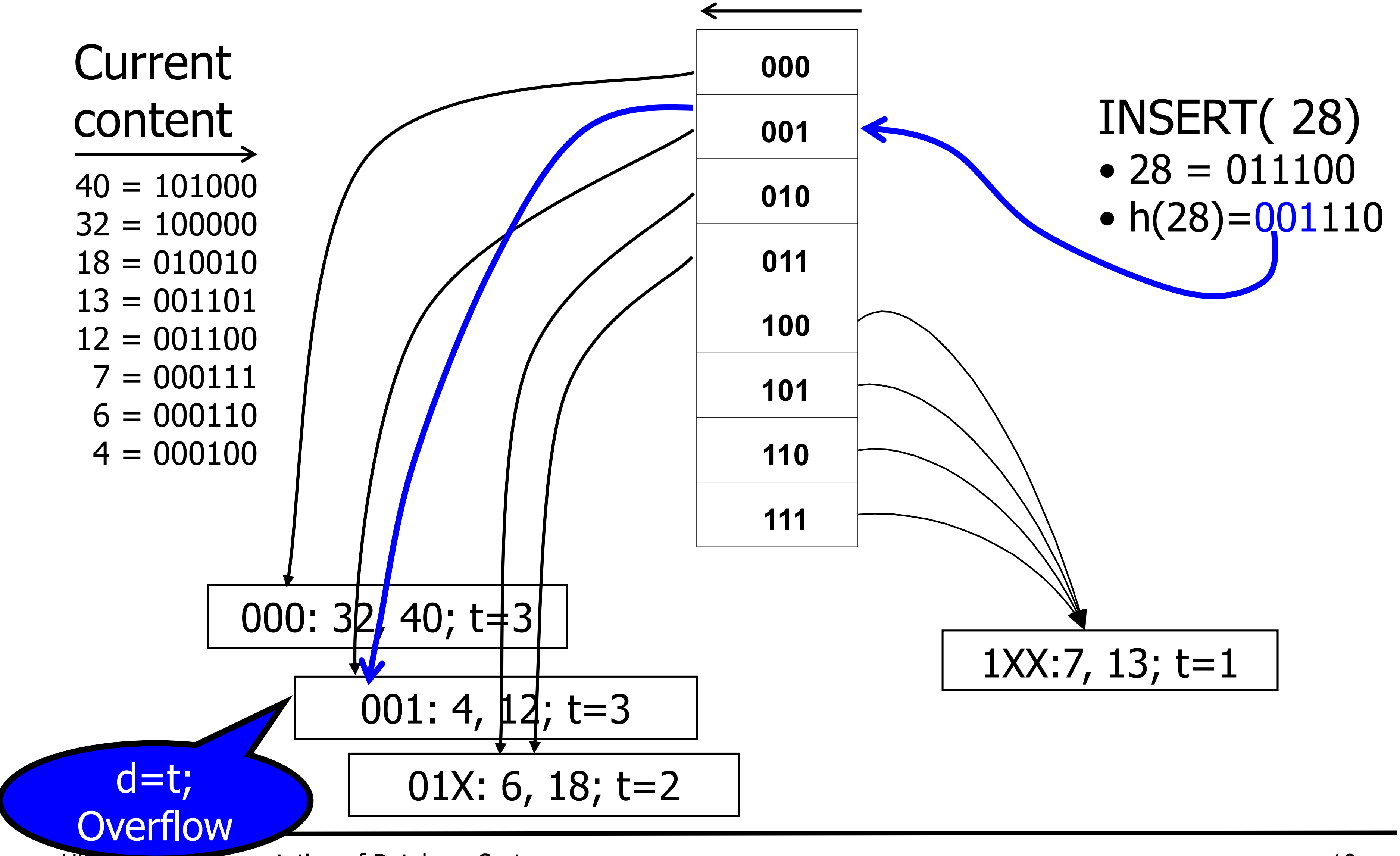
Example: Hash Table



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

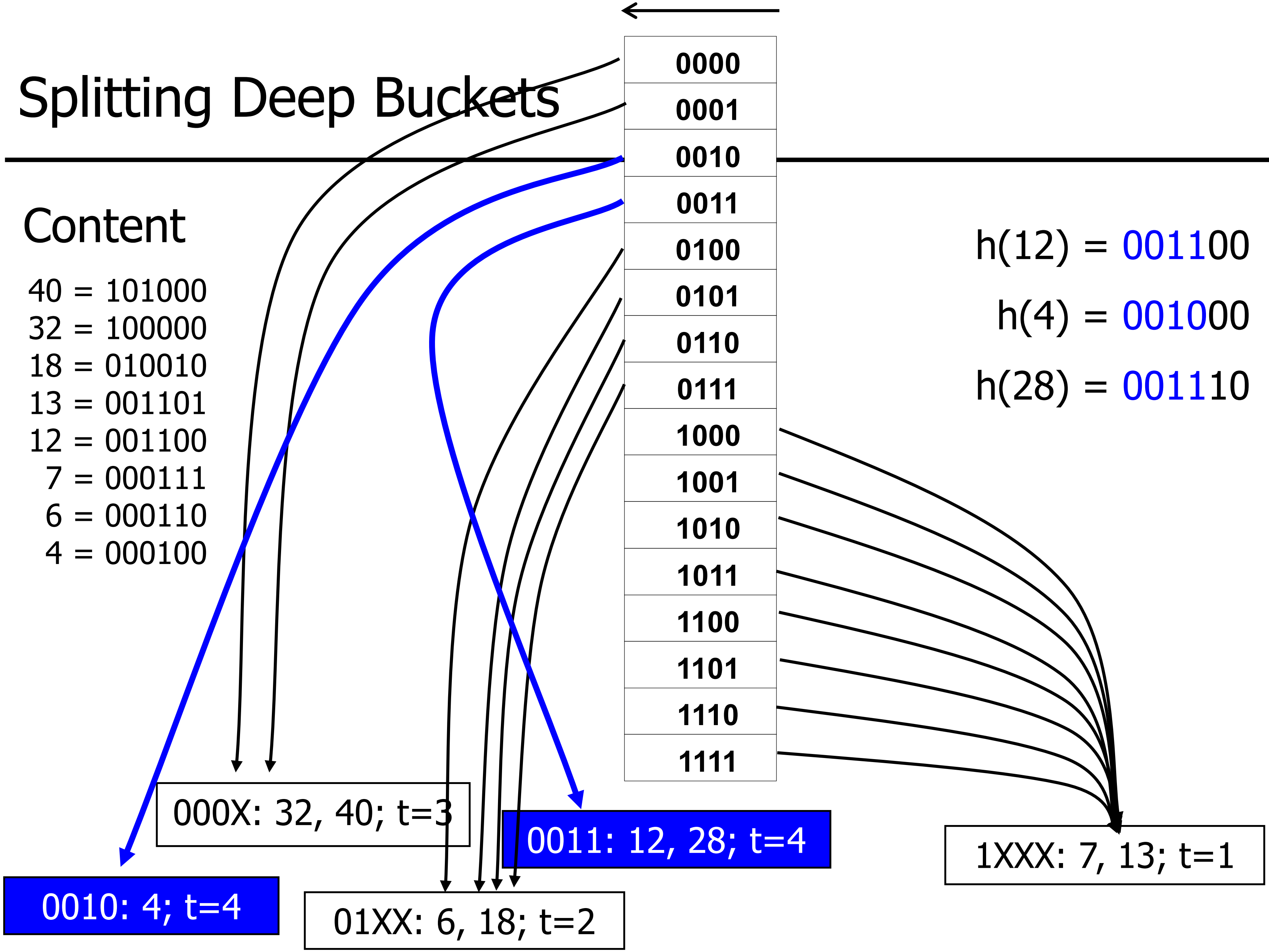
d=t; Overflow

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

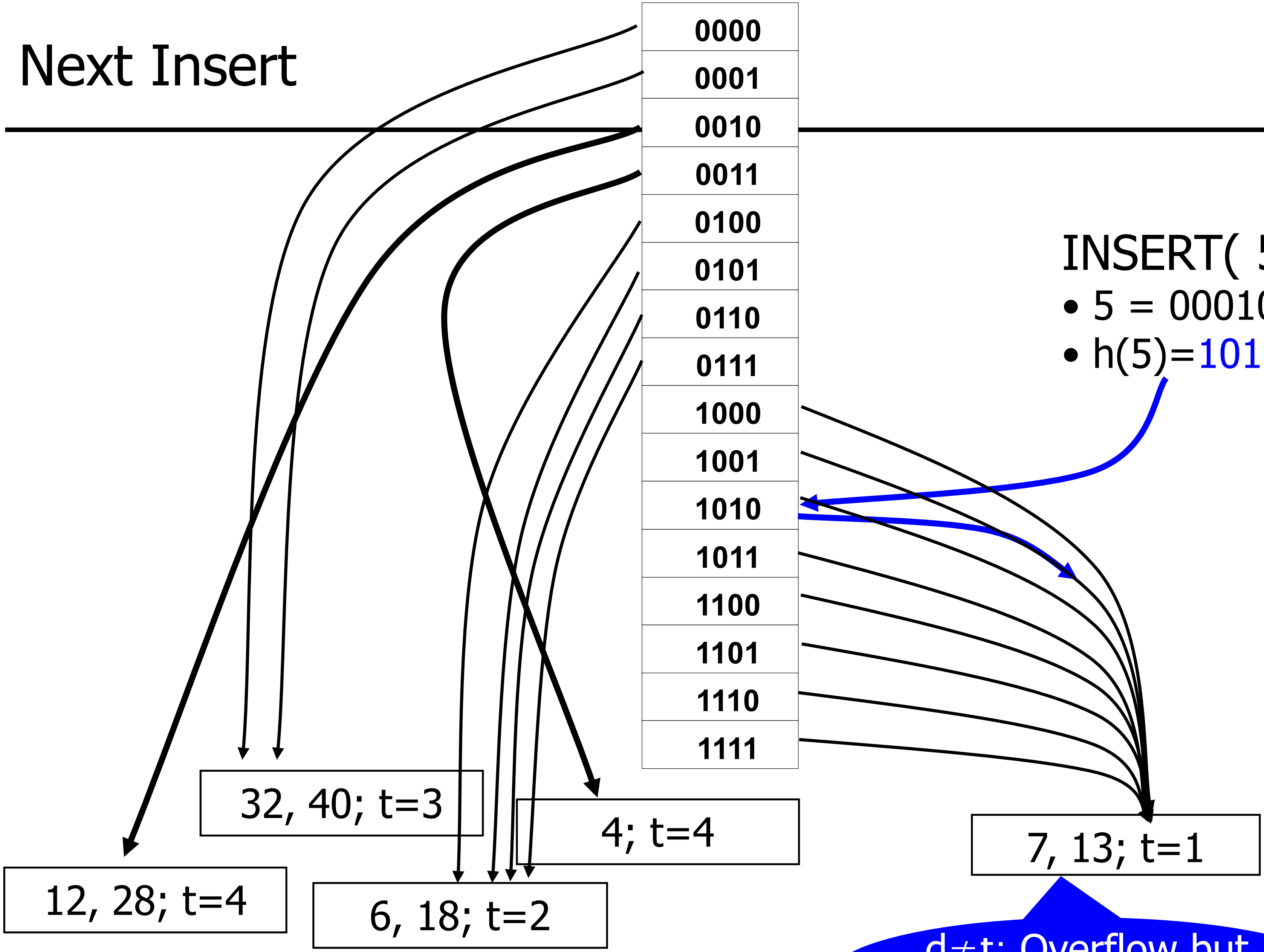
0011: 12, 28; t=4

1XXX: 7, 13; t=1

0010: 4; t=4

01XX: 6, 18; t=2

Next Insert



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 $r \in B$, 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
  - 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

Summary

- Advantages
  - Adapts to growing or shrinking number of records
    - Deletion not shown
  - 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

Content of this Lecture

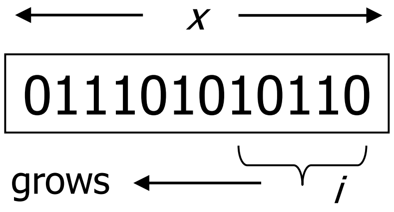
- Hashing
- Extensible Hashing
- Linear Hashing

Linear Hashing

- Similar to Extensible Hashing, but
  - Don’t double directory on overflow, but increase one-by-one
  - Guaranteed lower bound on bucket fill-degree
  - Leads to some overflow blocks in buckets
    - No more guarantee on 1 IO
    - But only little more if hash function spreads evenly

Overview

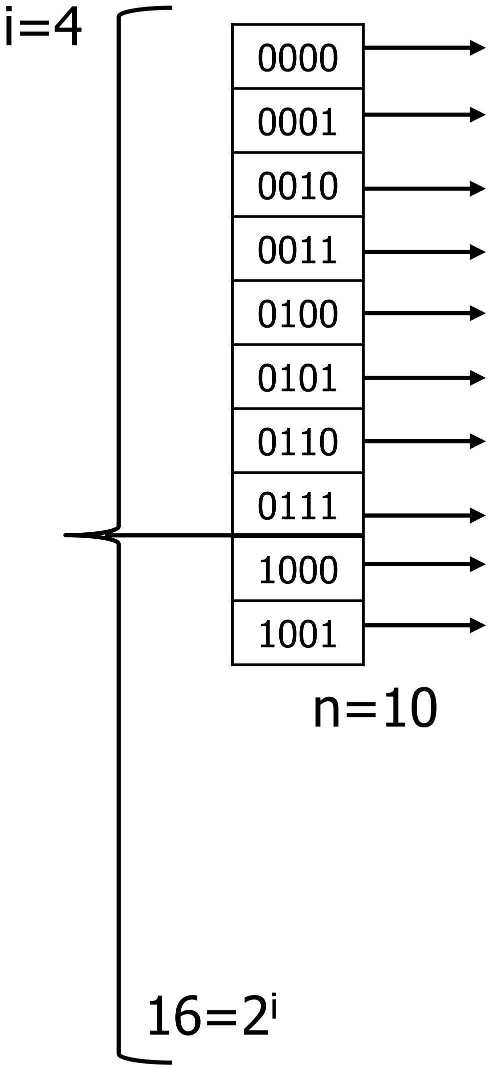
- h generates bitstring of length x, read right to left
- Parameters
  - i: Current number of bits from x used
    - As i grows, more bits are considered
    - If h generates x bits, we use $a_1a_2…a_i$ for the last i bits of h(k)
  - n: Total number of buckets currently used
    - Only the first n values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



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

Illustration

- We can address 2^i buckets
  - If we need more, i must be increased
- We actually have only n buckets
  - If we need more because of FDC, we need to increase n
  - As long as $n<2^i$ – no problem
  - Otherwise we first need to increase i
- A key k is hashed to a bistring h(k) whose last i bit are called m(k)
  - That is the address of k in the current hash table
  - m(k) maybe smaller than n (no problem or larger (problem)



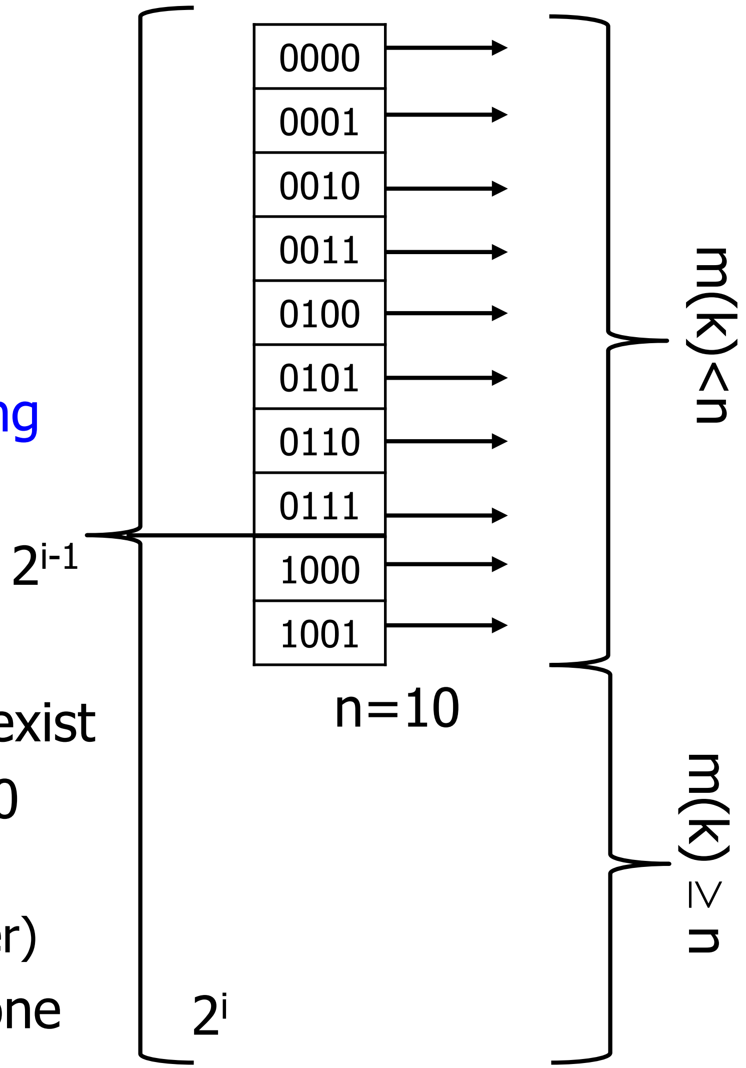
Inserting Overview

- Insert a new key k
- Has two phases

- 1^{st}: We store k
  - Compute m(k)
  - Bucket m(k) may exist or not; we insert anyway
- 2^{nd}: If FDC is hurt – repair
  - By inserting, r has grown by 1, so r/n might now be larger than t
  - We increase n (and possibly i)
  - This means creating a new bucket – where do we split?

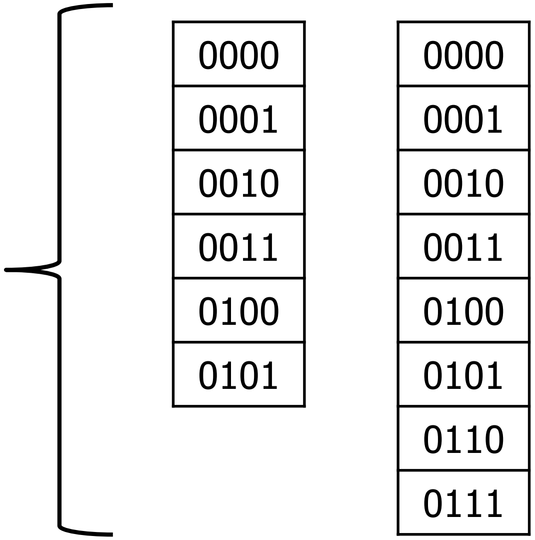
Insert(k): First Action

- Note: By construction, $n \geq 2^i$
  - Proof comes later
- If $m(k)<n$
  - The target bucket exists
  - Store k in bucket m(k), potentially using overflow blocks
- If $m(k) \geq n$
  - Bucket m(k) does not exist
  - We redirect k into a bucket that does exist
  - Flip i-th bit (from the right) of h(k) to 0 and store k in this bucket
    - By construction, this bit is 1 (proof later)
  - Note: 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 conceptual – no physical action
    - Proceed (now we have $n<2^i$)
  - If $n<2^i$
    - There is still room in our address space
    - We add (n+1)th bucket and set n++
    - Which bucket to split?
      - We do not split the bucket where we just inserted (why should we?)
      - We do not split the fullest bucket
      - Instead, we use a cyclic scheme (no extra admin cost)



0000	0000
0001	0001
0010	0010
0011	0011
0100	0100
0101	0101
	0110
	0111

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 $n=1a_2a_3\ldots a_i$; then we split block with ID $a_2a_3\ldots a_i$ into two blocks with ID $0a_2a_3\ldots a_i$ and ID $1a_2a_3\ldots a_i$
    - Requires redistribution of bucket with hash key $a_2a_3\ldots a_i$
    - This is one of the buckets where we had put redirected records
    - This is not necessarily an overflown 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

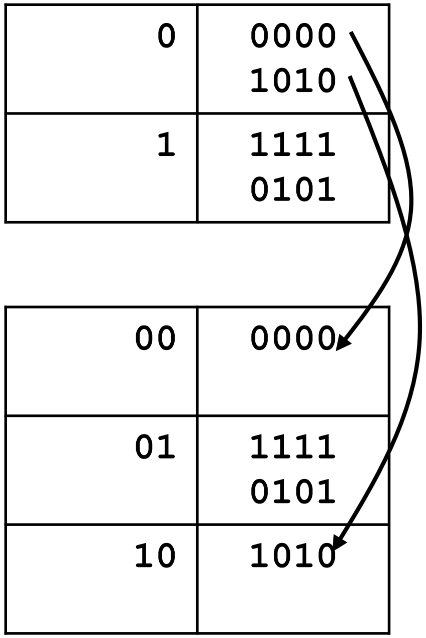
- Assume 2 records in one block, x=4, t=1.74, i=1

Start (with arbitrary keys)

0	0000 1010
1	1111

1a) Insert k=0101
m(k)=1<n=2
Insert into bucket 1
But now r/n≥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++



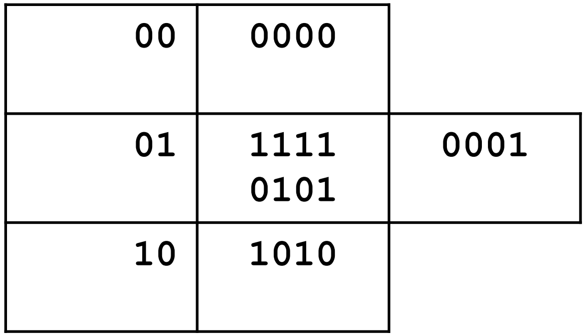
0	0000 1010
1	1111 0101

00	0000
01	1111 0101
10	1010

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

Example 2

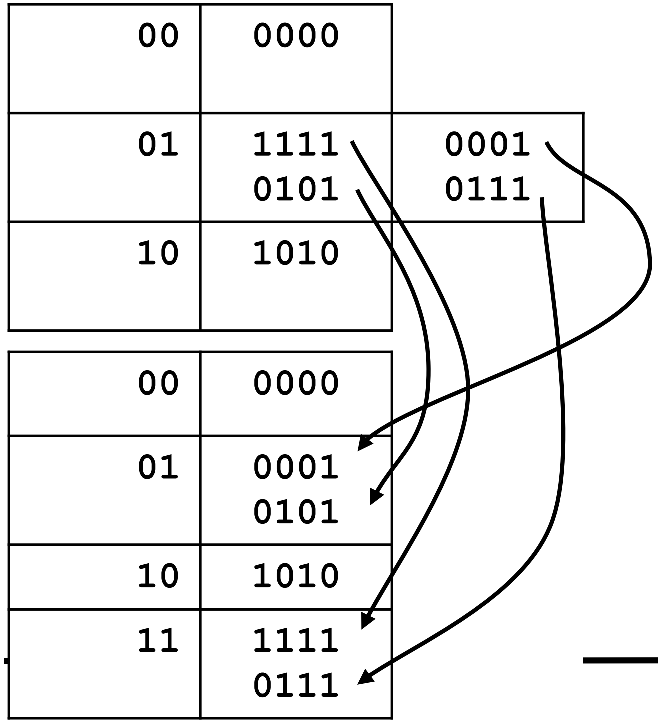
2) Insert $k=0001$
$m(k)=1$, bucket exists
Insert into $m(k)$
Requires overflow block



00	0000	
01	1111 0101	0001
10	1010	

3a) Insert $k=0111$
$m(k)=3=n=11_b$
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 $3=11_b$
Since $n=11_b$, we split 01
Removes (here) overflown block



00	0000	
01	1111 0101	0001 0111
10	1010	

00	0000
01	0001 0101
10	1010
11	1111 0111

Example 3

4a) Insert 0011

$m(k)=3=11_b < n=4=100_b$

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 overflown

Observations (Proofs)

- 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 $1b_1b_2...b_{i-1}$
- By definition: $m(k)<2^i$
  - But possibly: m>n
    - Such m must have a leading 1, as n must have one (see previous observation)
    - If we drop the leading 1 in m, we get $m_{new}<2^{i-1}$
    - Since $n \geq 2^{i-1}$, $m_{new} \leq n$
    - Thus, the chosen bucket m_{new} must already exist
- How do we implement the hash table?
  - Not as array, as it must grow (and shrink)
  - Linked list (linear search in memory) or AVL tree (log(n))

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-access IO on disk through overflow blocks