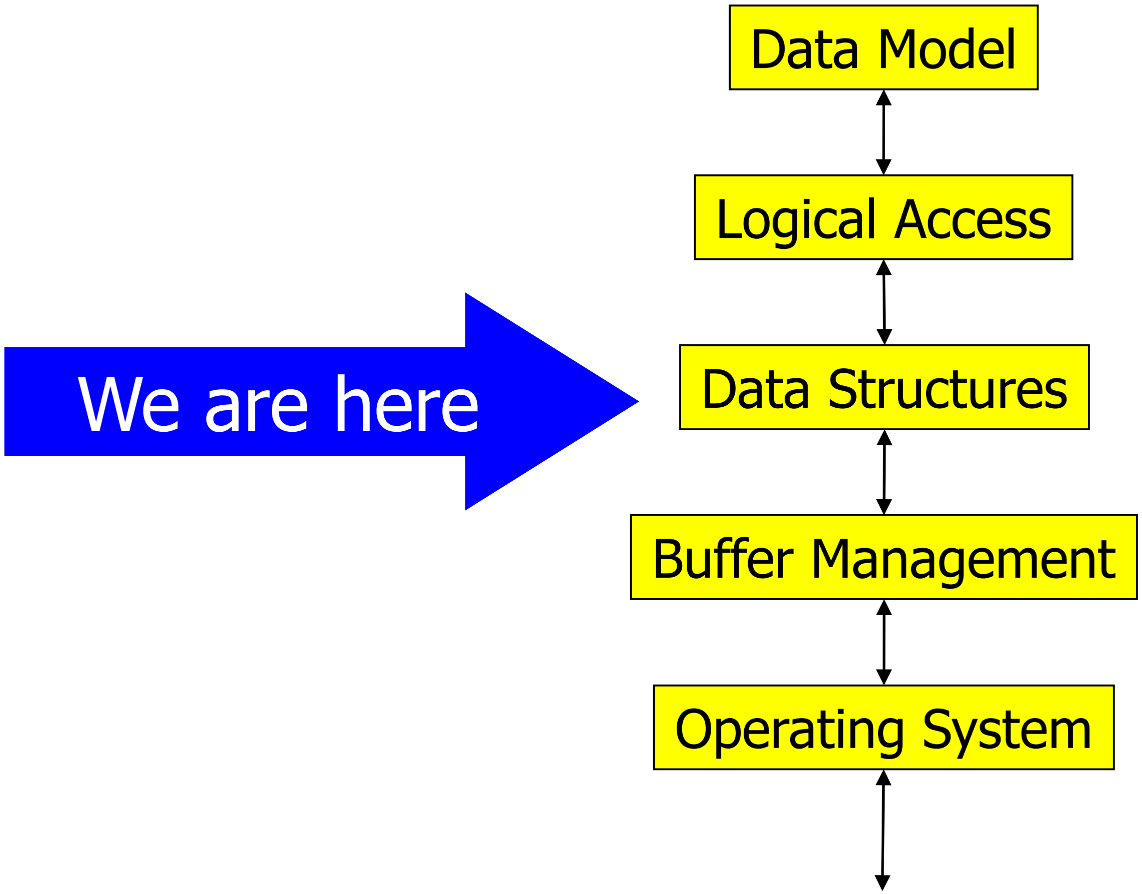
Datenbanksysteme II: Dynamic Hashing

Ulf Leser

5 Layer Architecture



Content of this Lecture

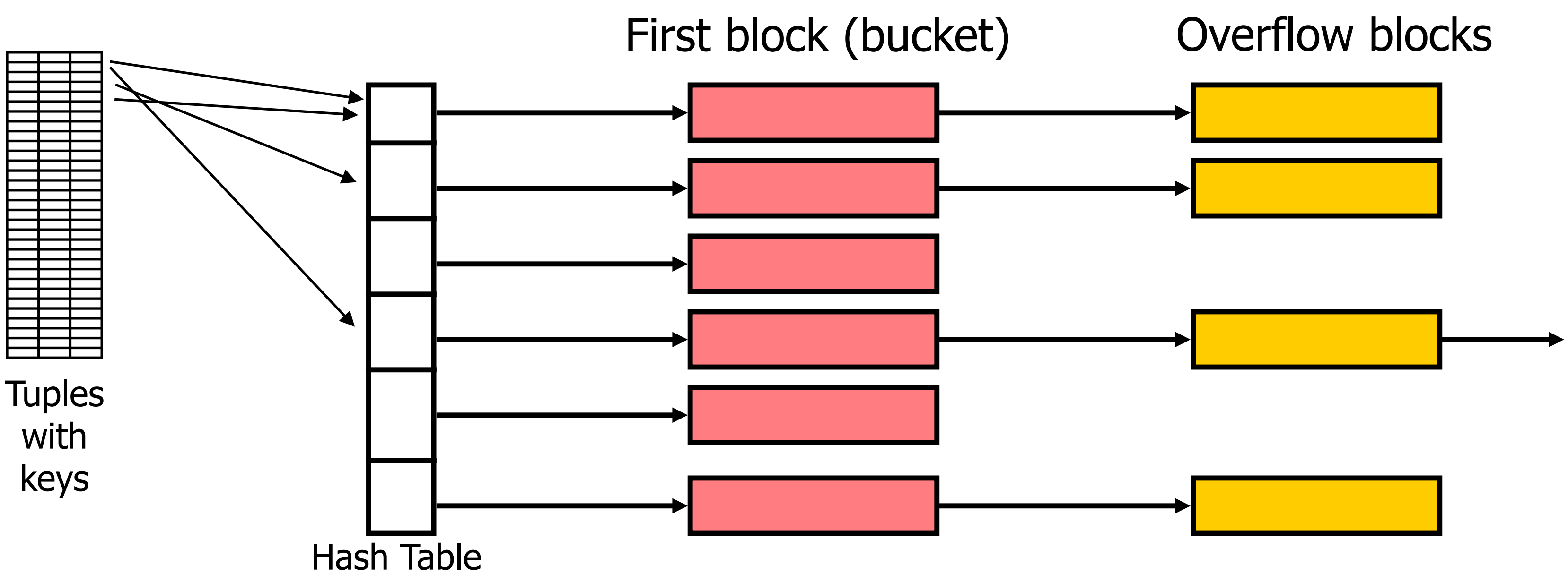
- Hashing
- Extensible Hashing
- Linear Hashing

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 with 1 IO
    - Searching for multiple keys – multiple hash indexes
  - Incurs overhead if table size changes considerably
    - Dynamic hashing
  - 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(key) = {0 ,..., m-1}
- Hash table H (bucket directory) of size m with ptrs to B_i's

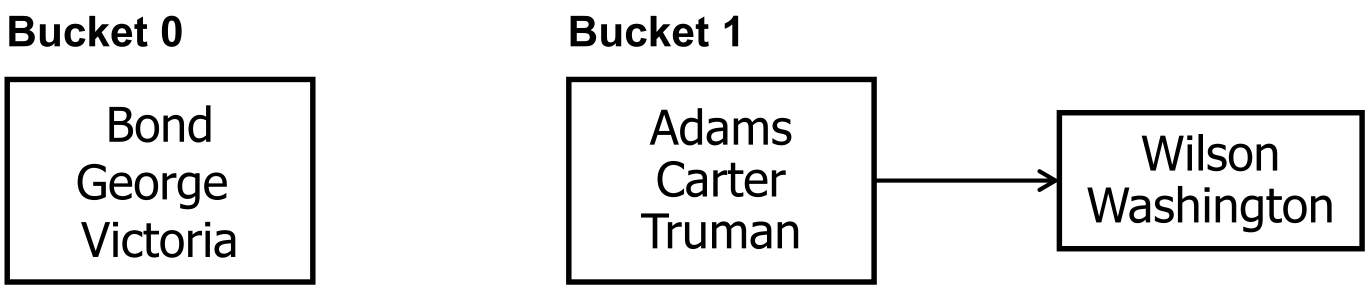


Example

- 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?



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

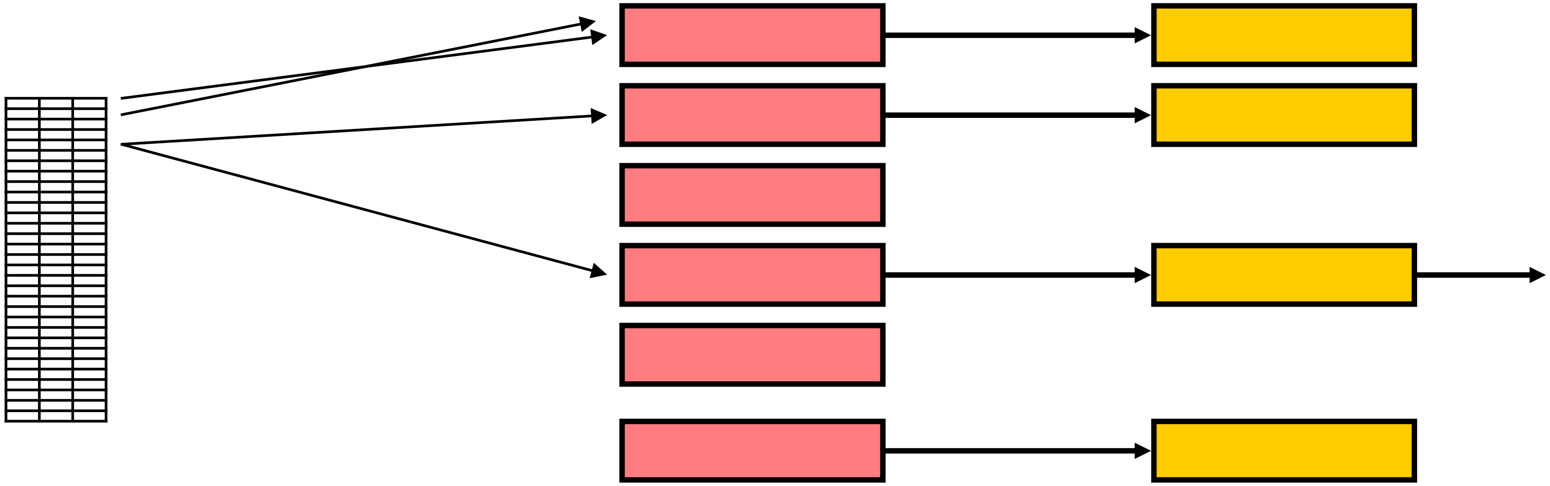
Search "Swift"

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

Failure

Alternative: Direct Block Hashing

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



Efficiency of Hashing

- Given n records, r records per block, m buckets
- Assume hash table H is in main memory
- Average number of blocks per bucket: n / (m*r)
  - Assuming a uniform hash function and no empty space
  - Difficult to achieve in practice
- Search (block reads)
  - n / (m*r) / 2 for successful search
  - n / (m*r) for unsuccessful search (entire bucket)
- Insert
  - n / (m*r) if end of bucket cannot be accessed directly
  - depends… 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
- 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 hash table requires complete rehashing
  - We need a new hash function
  - Table lock: Blocks all operations on this table
- Inefficient for range queries – scan
  - Or enumerate all distinct values in range (only integer)
- Very fast iff everything works fine
  - "Practically constant" IO complexity

Very bad for growing tables = for databases

Content of this Lecture

- Hashing
- Extensible Hashing
- Linear Hashing

Extensible Hashing

- For DBMS, hashing must adapt to changing data volumes and value distributions
  - Dynamic hashing
- First idea: Extensible Hashing
  - Hash function generates (long) bitstrings
    - 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
  - Overflows requires rehashing table only partly

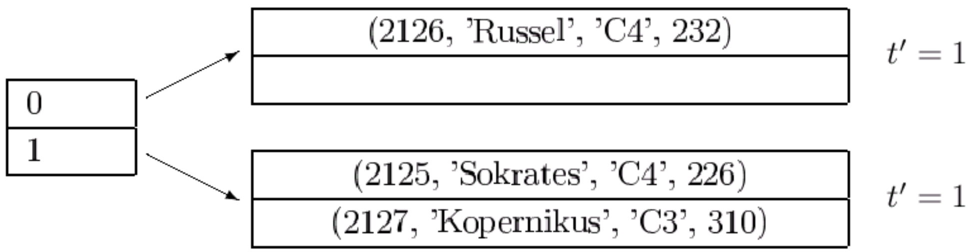
Hash functions

- h: K $\rightarrow$ {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: reverse 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

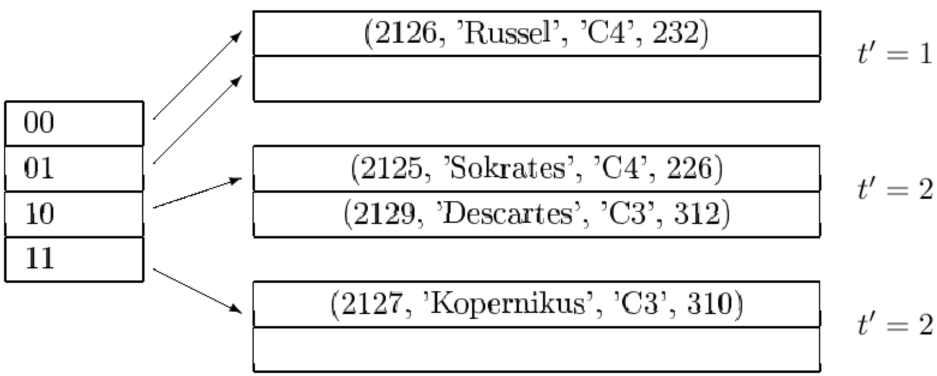
- Parameters
  - d: global „depth" of hash table, size of longest prefix currently used
  - t: local „depth" of a bucket, size of prefix used in this bucket
- Example
  - Let a bucket store two records
  - Start with two buckets and 1 bit for identification ($d=t_1=t_2=1$)

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



Example cont'd

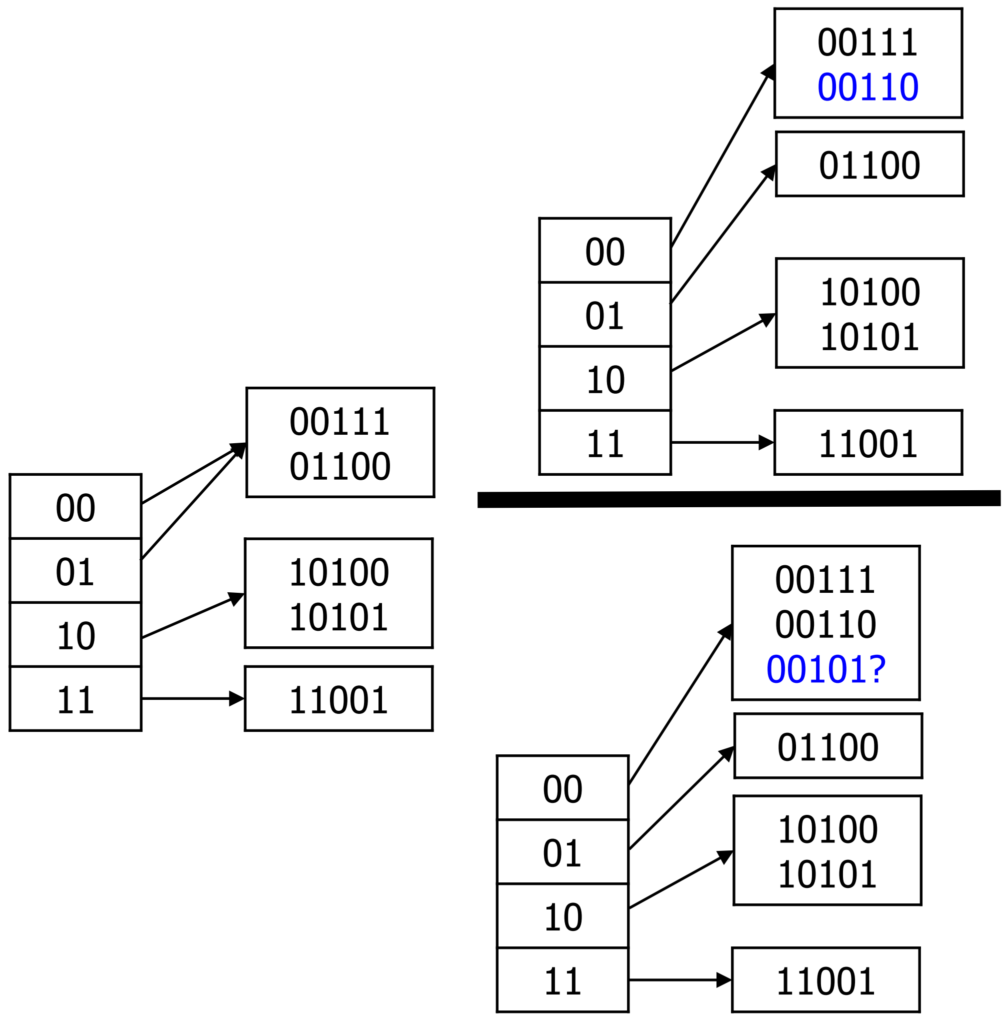
k	as bitstring	reverse	$h_{d=1}$
2125	100001001101	101100100001	1
2126	100001001110	011100100001	0
2127	100001001111	111100100001	1
2129	100001010001	100010100001	1



- 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

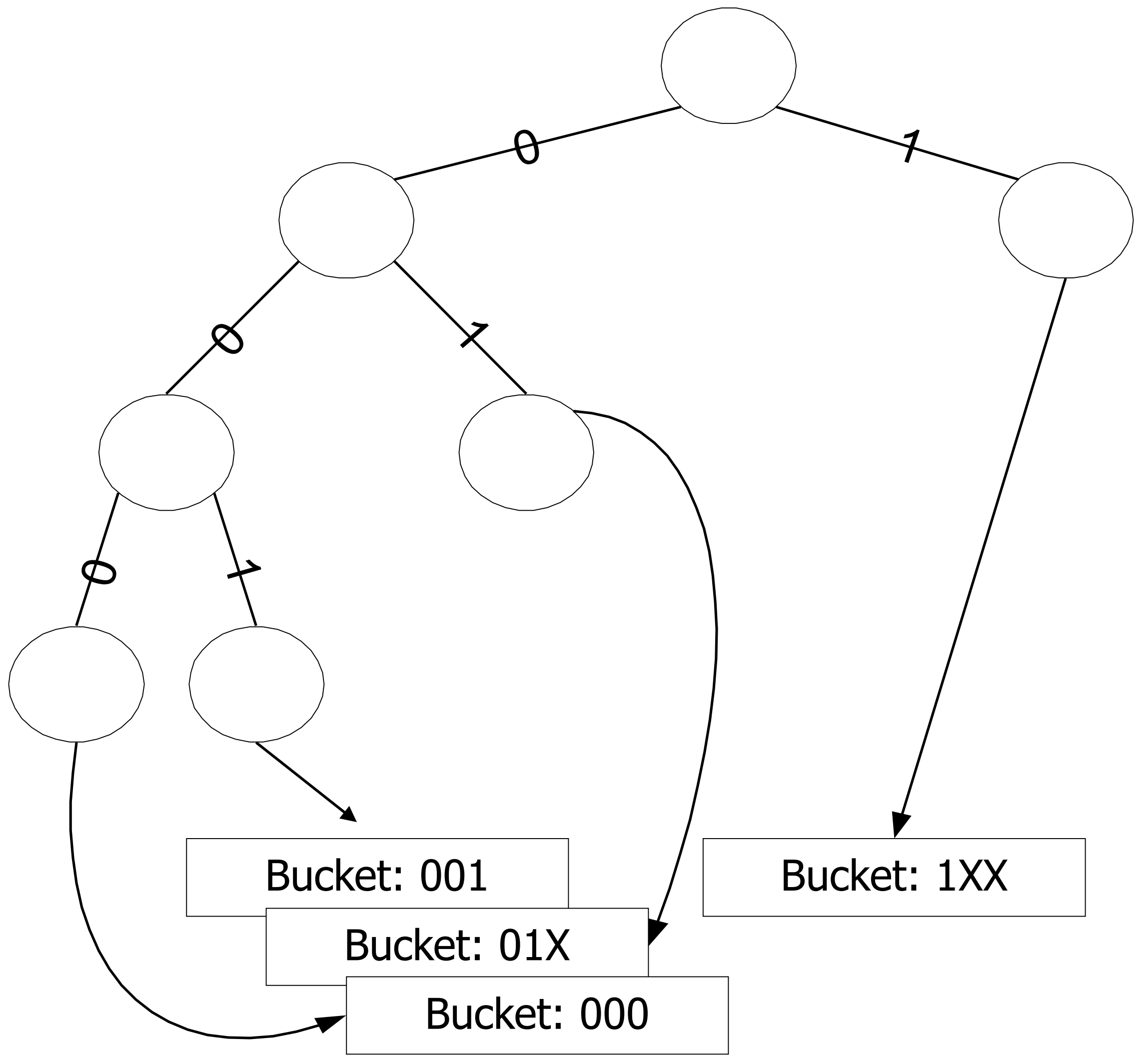
Special Cases

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

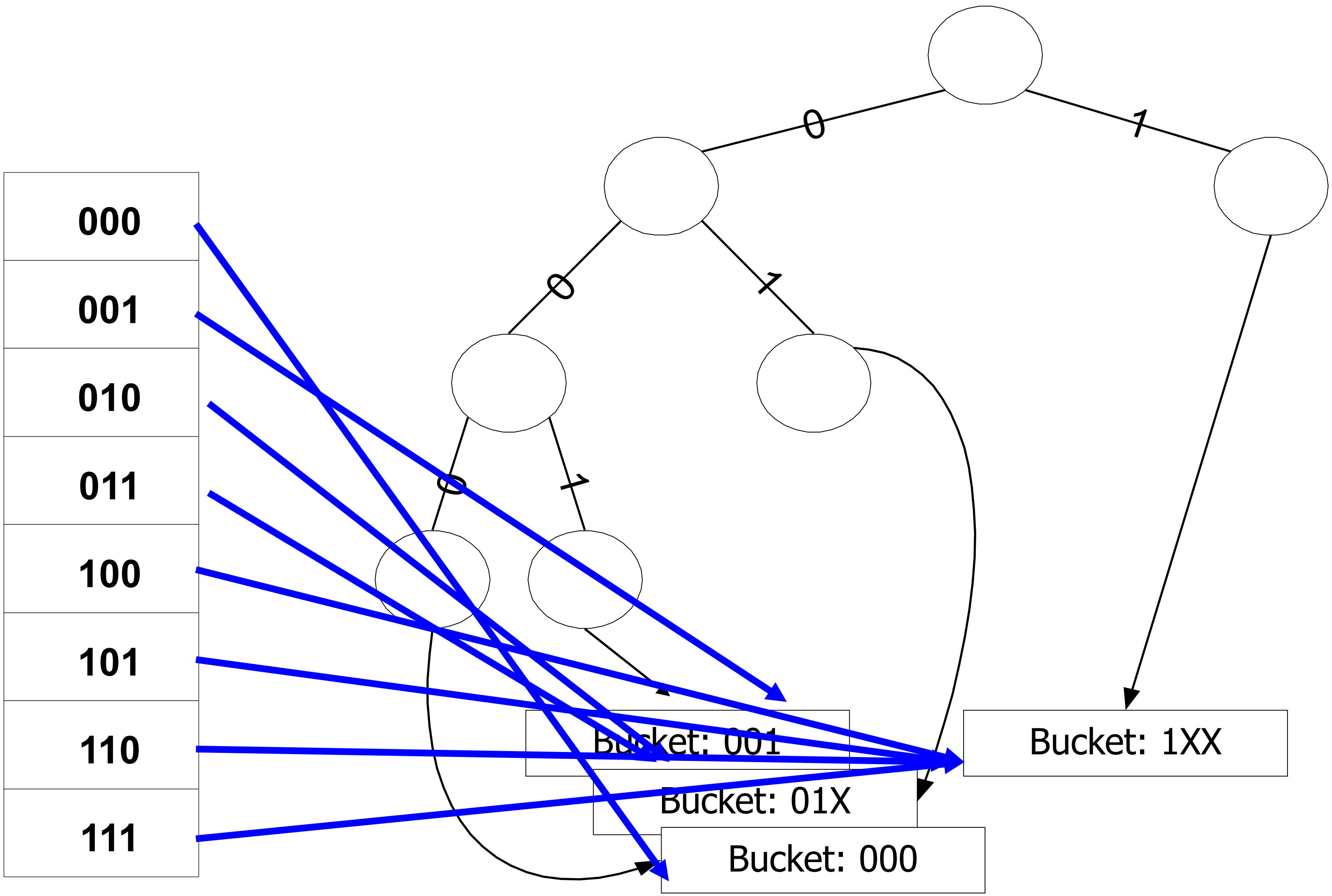


More Complex Example

- 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 hash table: $2^d=8$



Example: Hash Table



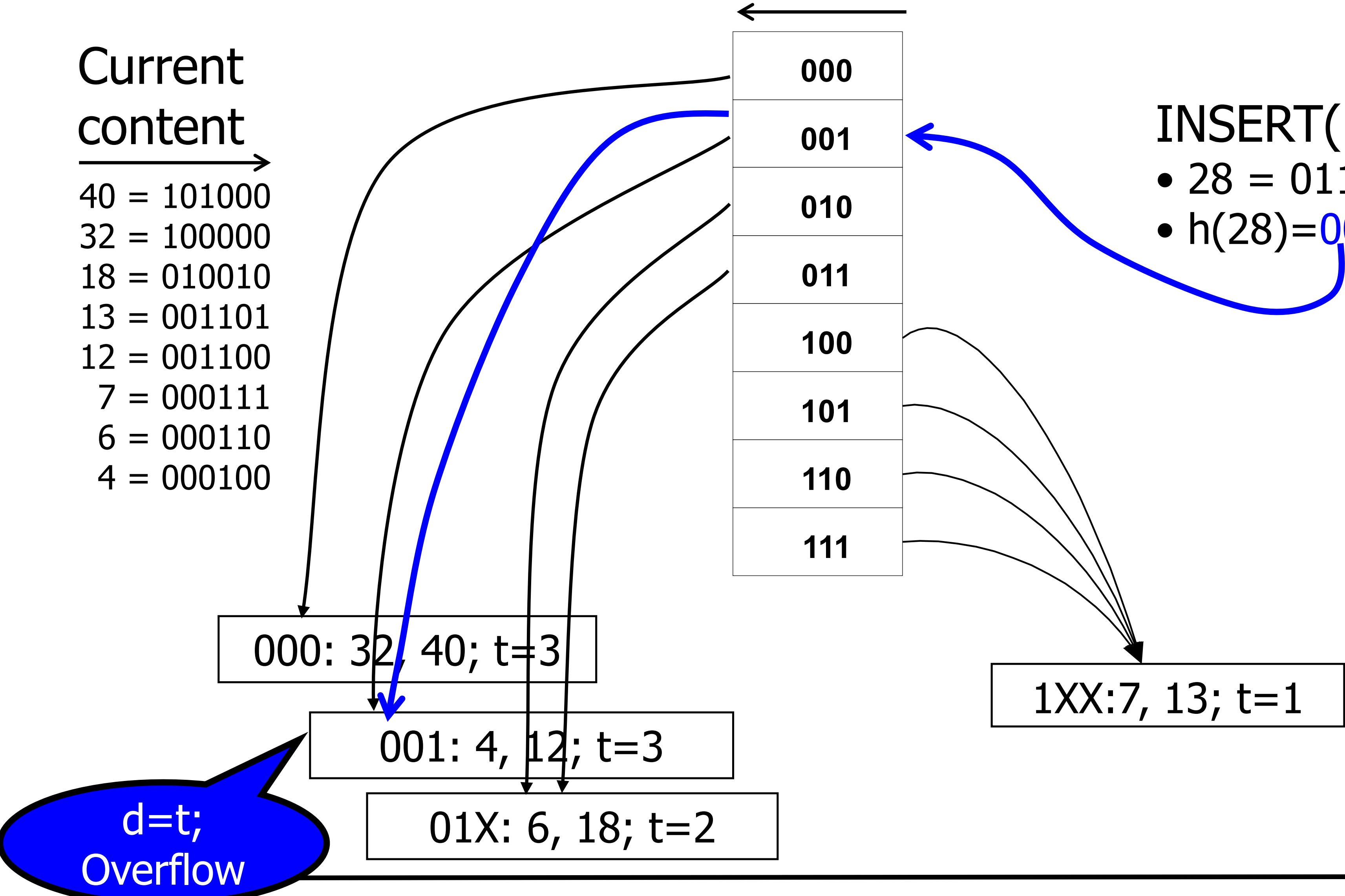
Inserting Values

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

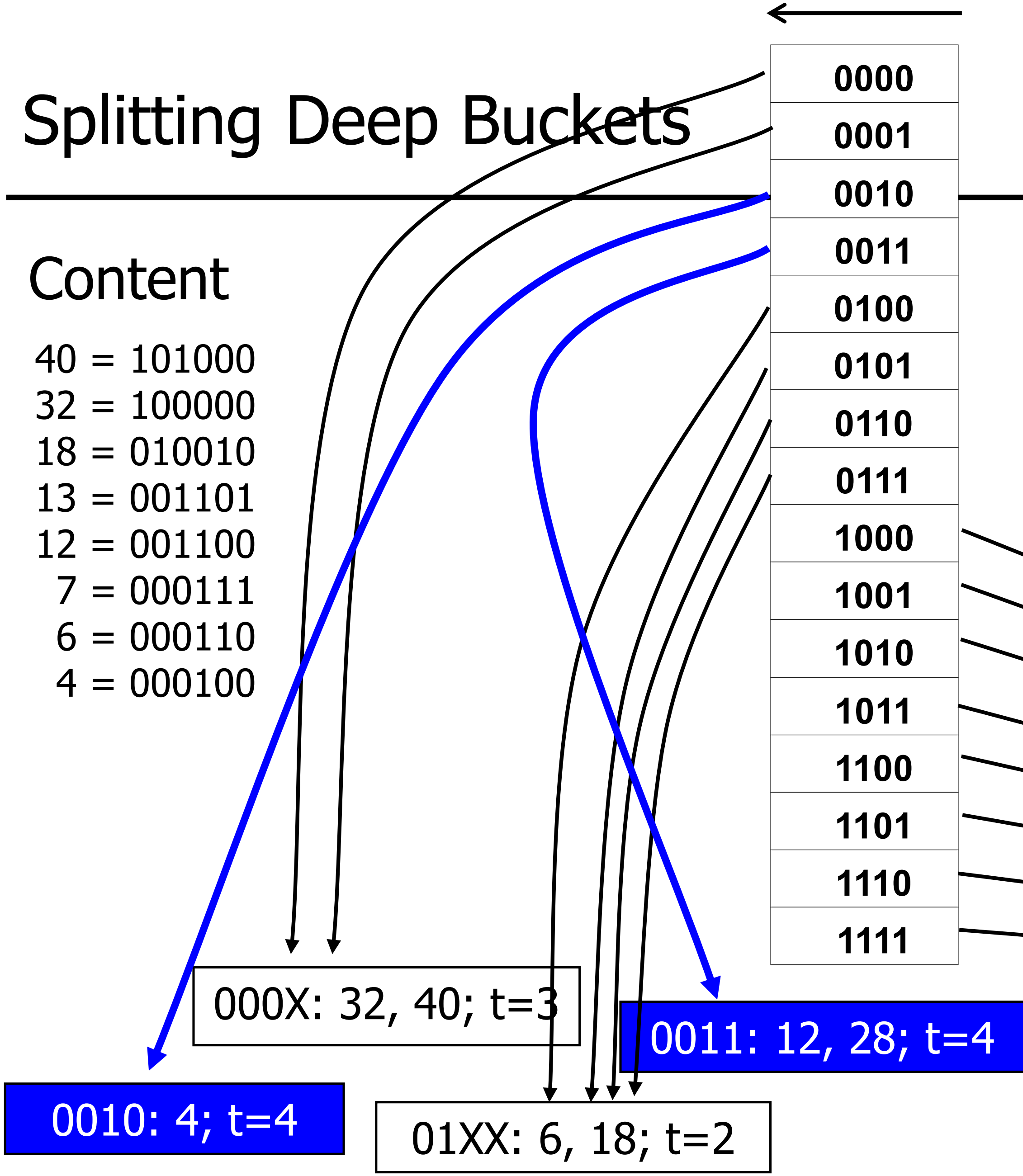
Content

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

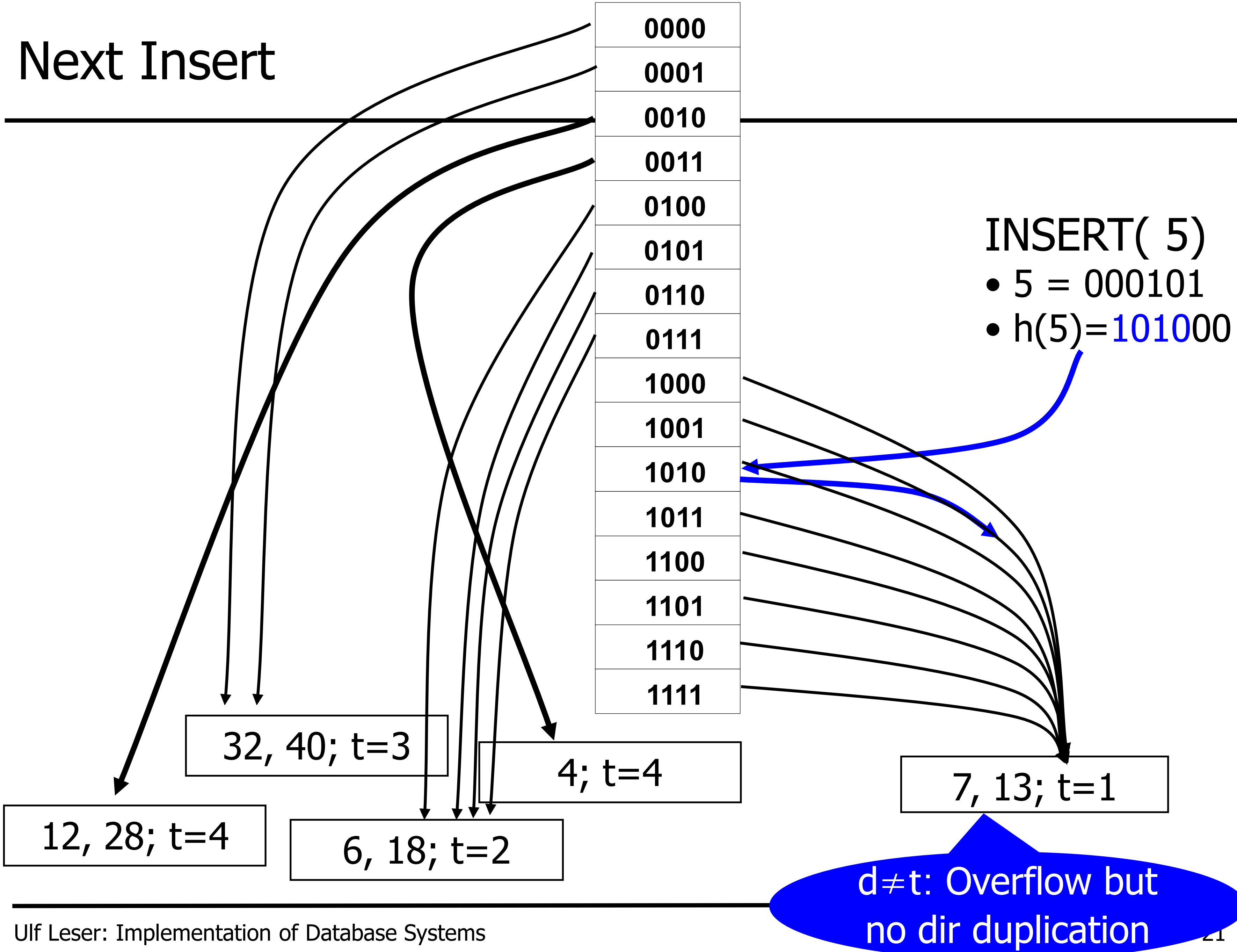
h(12) = 001100

h(4) = 001000

h(28) = 001110



Next Insert


0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
INSERT(5)
• 5 = 000101
• h(5)=101000
32, 40; t=3
4; t=4
7, 13; t=1
12, 28; t=4
6, 18; t=2
d≠t: Overflow but no dir duplication

Splitting Shallow Buckets

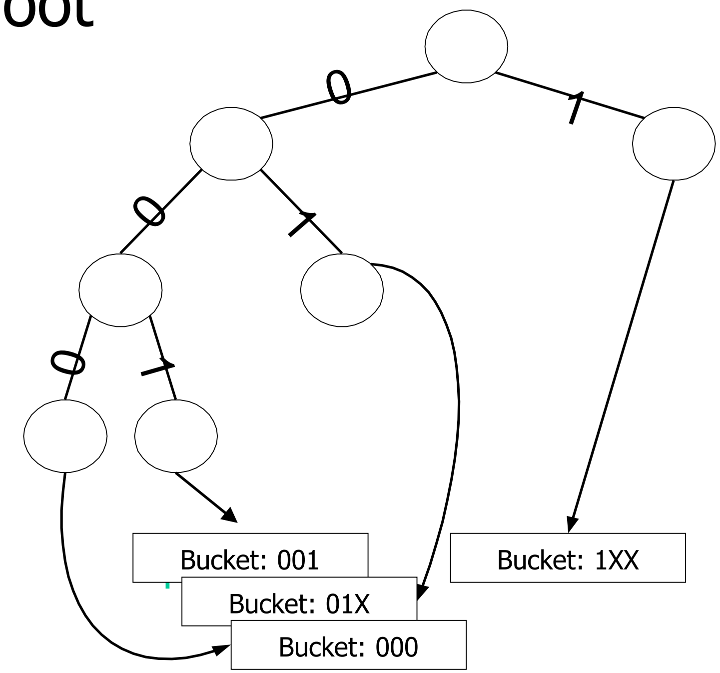
- Assume we have to split overflowing bucket B
- B is shallow if t<d
- For all records r∈B, h(r) has the same length-t prefix
- If we split at next position (t++)
  - Generate new bucket and rehash records
  - This might generate an empty bucket
  - The other bucket might still be overflowing – repeat 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
  - Deep split: Hash table doubles

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

Exponentially Growing Hash Table?

- Can be avoided
- Organize hash table as tree
- Ranges of buckets with local depth smaller than global depth are leaves closer to root
- Properties
  - May drastically reduce memory requirements
  - Access is slower: Following pointers, random access in main memory



Content of this Lecture

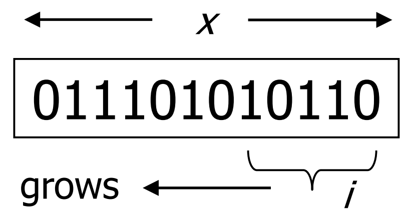
- Hashing
- Extensible Hashing
- Linear Hashing

Linear Hashing

- Similar to Extensible Hashing, but
  - Doesn't double directory on overflow, but increases site one-by-one
  - Guaranteed lower bound for 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

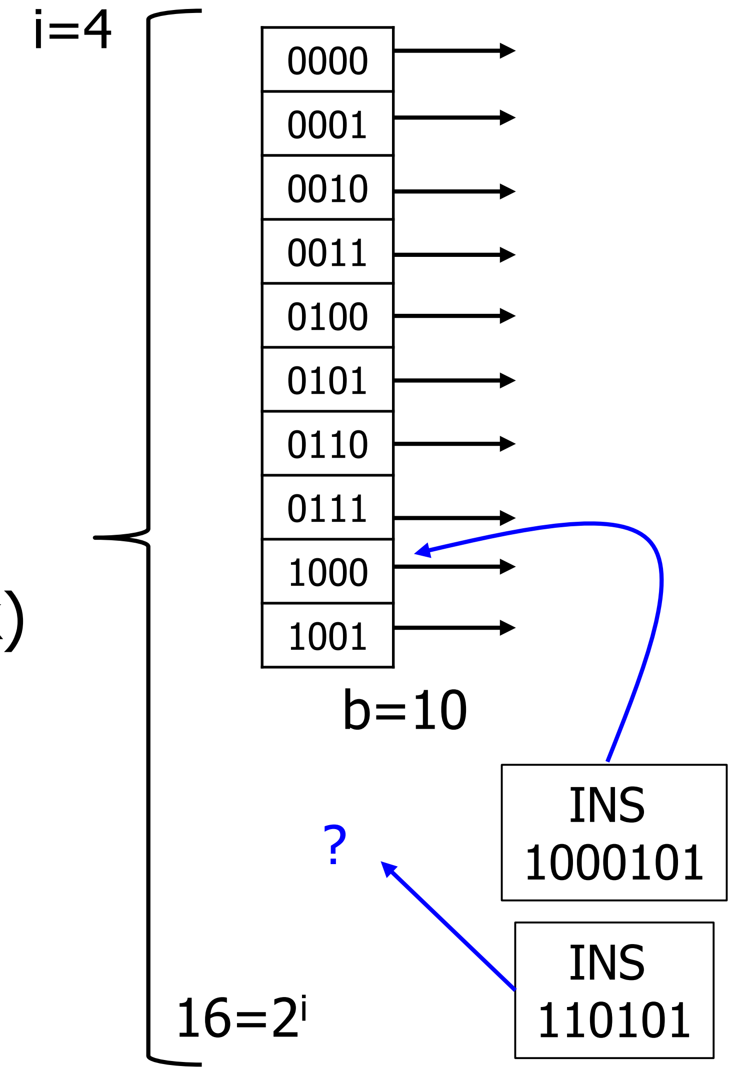
- h generates bitstring of length x, read right to left
- Global parameters



  - i: Current number of bits from x used
    - As i grows, more bits are considered
    - If h generates x bits, we use $a_1a_2 \ldots a_i$ for the last i bits of h(k)
  - b: Total number of buckets currently used
    - Only the first b values of bitstrings of length i have their own buckets
  - n: Total number of records
- Fix threshold t – linear hashing guarantees that $n/b<t$
  - The fill-degree constraint (FDC)
  - As n increases, we sometimes must increase b to keep FDC
  - 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, but not the WC

Illustration

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



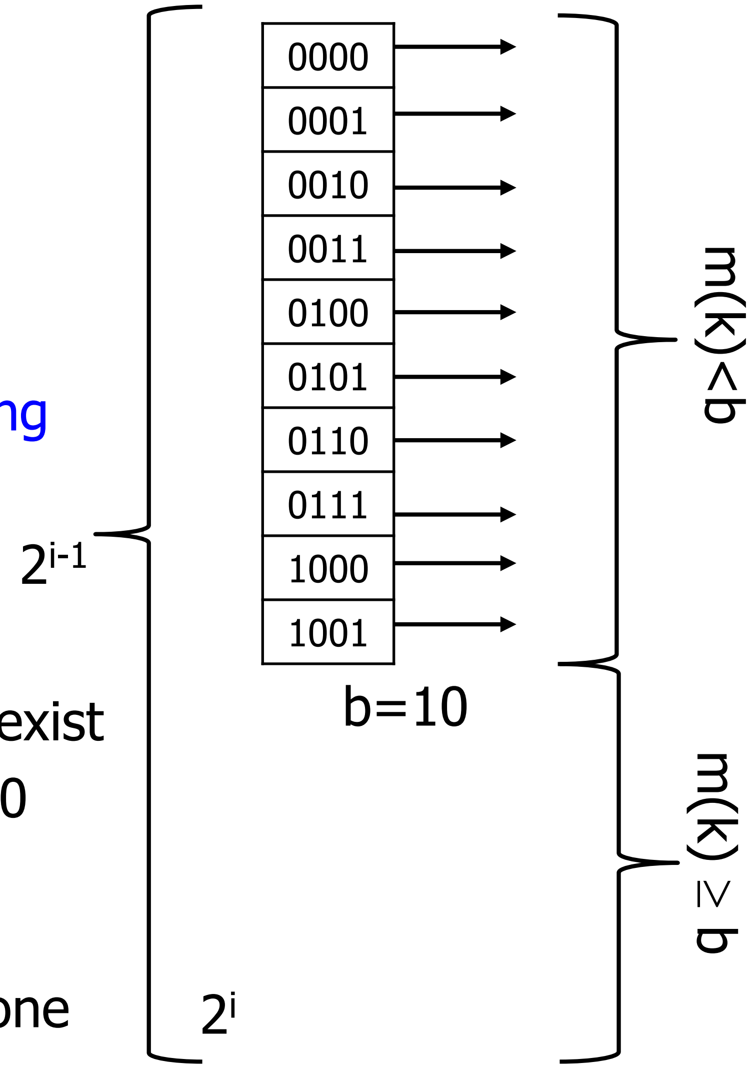
Inserting Overview

- Inserting a new key k has two phases

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

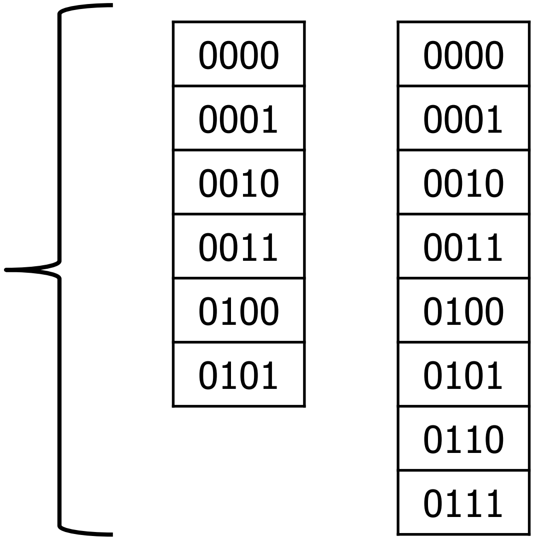
Insert(k): First Action

- Note: By construction, $b \geq 2^{i-1}$
  - Proof comes later
- If $m(k) < b$
  - The target bucket exists
  - Store k in bucket m(k), potentially using overflow blocks
- If $m(k) \geq b$
  - 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
    - This bit is 1 (proof later)
  - Note: This flipping also needs to be done when searching keys



Insert(k): Second Action

- Check threshold; if $n/b \geq t$, then
  - If $b=2^i$
    - No more room to add another bucket
    - Set i++
    - This is only conceptual – no physical action
    - Proceed (now we have $b<2^i$)
  - If $b<2^i$
    - There is still room in our address space
    - We add (b+1)th bucket and set b++
    - Which bucket to split?
      - We do not split the bucket where we just inserted
      - We do not split the fullest bucket
      - Instead, we use a cyclic scheme to avoid extra admin cost



Which Bucket to Split

- We split buckets in fixed, cyclic order
- Always split bucket with number $b-2^{i-1}$
  - As b increases, this cycles through all buckets
  - Let $b=1a_2a_3...a_i$; 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 overflown bucket
    - Recall: Only the average fill degree is guaranteed

Buckets Split Order

Assume we would split after every insert

i	b	Existing buckets	Bucket to split: $b-2^{i-1}$	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, bitstring length x=4, t=1.74, i=1

0	0000 1010
1	1111

Start situation

1a) Insert k=0101
m(k)=1<b=2
Insert into bucket 1
But now $n/b \geq t$

0	0000 1010
1	1111 0101

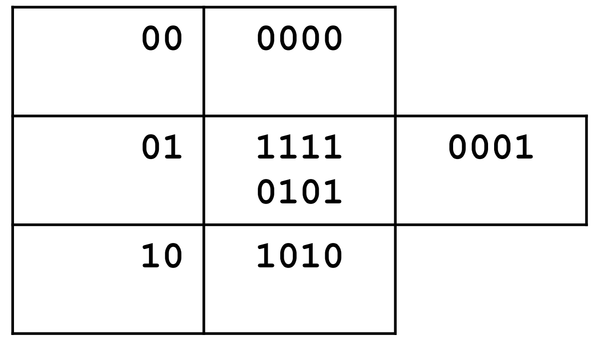
1b) Since $b=2^i=2=10_b$
We need more address space
Increase i (virtually)
Add bucket number $2=10_b$
$b=10_b=1a_1$: Split bucket 0 into 10 and 00
b++

00	0000
01	1111 0101
10	1010

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

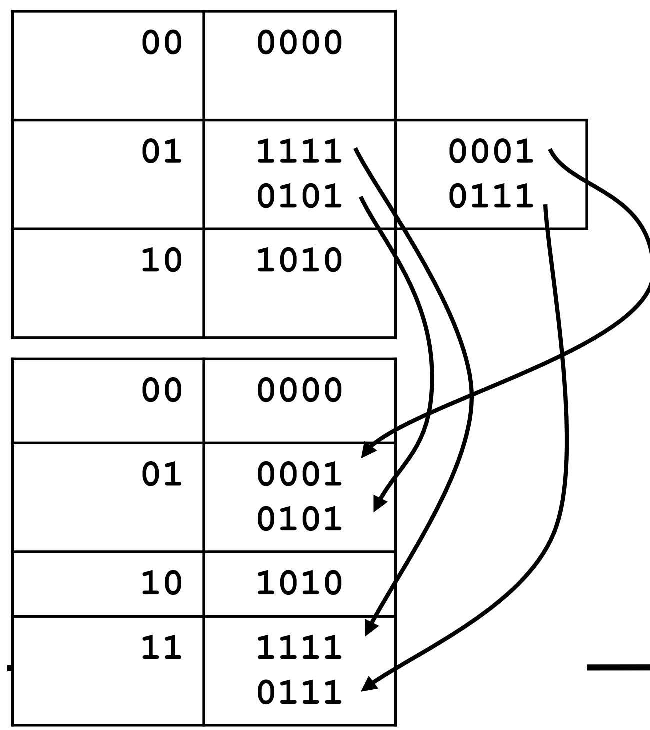
Example 2

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



3a) Insert k=0111
$m(k)=3=b=11_b$
Bucket doesn't exist
Flip and redirect to 01

3b) Now $n/b=6/3 \geq t$ – We split
b<4, so no need to increase i
Add bucket number $3=11_b$
Since $b=11_b$, we split 01
Removes (here) overflown block



Example 3

4a) Insert 0011
  $m(k)=3=11_b < b=4=100_b$
  Insert into 11_b

00	0000	
01	0001 0101	
10	1010	
11	1111 0111	0011

4b) We must split again
  Since $b=2^i$, 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: $2^{i-1} \leq b \leq 2^i$
  - Whenever b reaches 2^i, i is increased => 2^i doubles and $b=2^i/2$ (for the new i)
  - Hence, b as binary number always has the form $1b_1b_2...b_{i-1}$
- By definition: $m(k)<2^i$
  - But possibly: m>b
    - Such m must have a leading 1, as b must have one (see previous observation)
    - If we drop the leading 1 in m, we get $m_{new}<2^{i-1}$
    - Since $n \geq 2^{i-1}$, $m_{new} \leq b$
    - Thus, the chosen bucket m_{new} must already exist
- How do we implement the hash table?
  - Not as array, as it must grow in small steps (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
  - Guaranteed fill degree
- Disadvantages
  - Search can degrade, as buckets are split in fixed order
  - No adaptation to skewed value distribution
  - Creates random-access IO on disk through overflow blocks