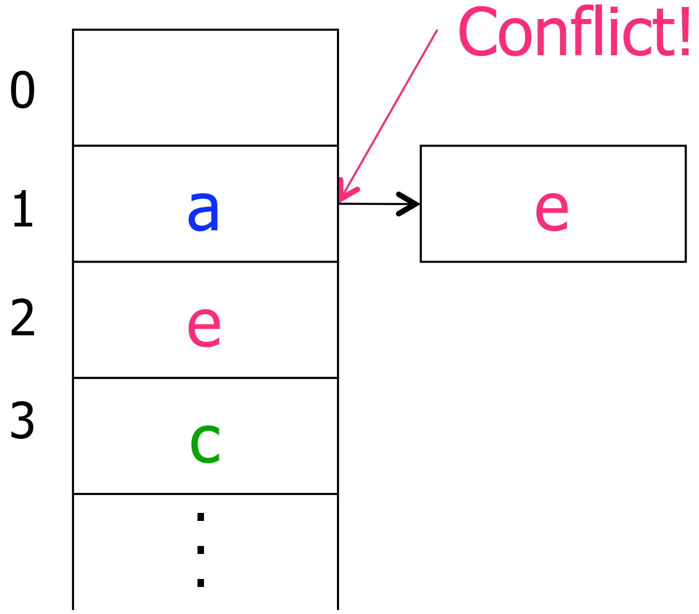
Hash Tables

Hash Table in Primary Storage

- Main parameter B = number of buckets
- Hash function h maps key to numbers from 0 to B-1
- Bucket array indexed from 0 to B-1
- Each bucket contains exactly one value
- Strategy for handling conflicts

Example: B = 4

- Insert c (h(c) = 3)
- Insert a (h(a) = 1)
- Insert e (h(e) = 1)
- Alternative 1:
  - Search for free bucket, e.g. by Linear Probing
- Alternative 2:
  - Add overflow bucket



Hash Function

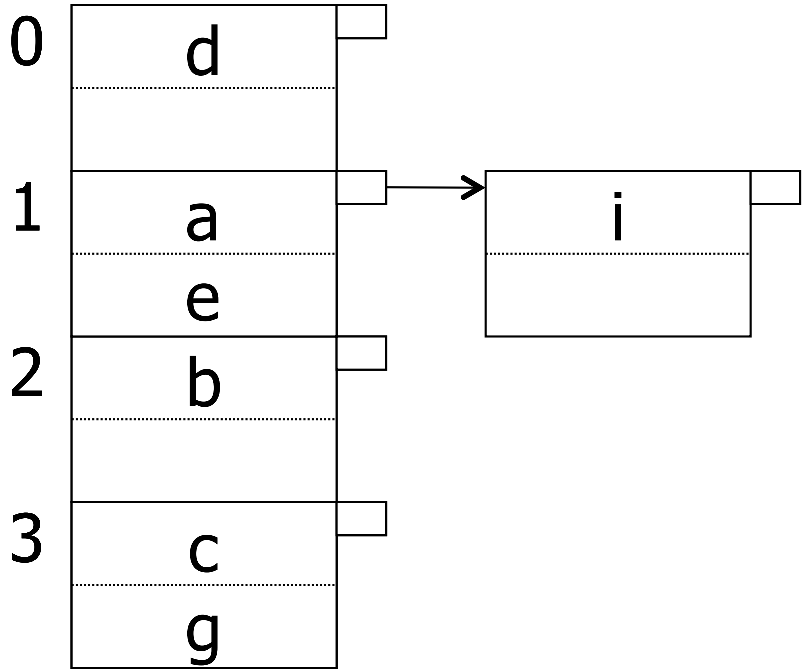
- Hash function should ensure hash values are equally distributed
- For integer key K, take h(K) = K modulo B
- For string key, add up the numeric values of the characters and compute the remainder modulo B
- For really good hash functions, see *Donald Knuth, The Art of Computer Programming: Volume 3 – Sorting and Searching*

Hash Table in Secondary Storage

- Each bucket is a block contains *f* key-pointer pairs
- Conflict resolution by probing potentially leads to a large number of I/Os
- Thus, conflict resolution by adding overflow buckets
- Need to ensure we can directly access bucket *i* given number *i*

Example: Insertion, B=4, f=2

- Insert a
- Insert b
- Insert c
- Insert d
- Insert e
- Insert g
- Insert i



Efficiency

- Very efficient if buckets use only one block: one I/O per lookup
- Space utilization is #keys in hash divided by total #keys that fit
- Try to keep between 50% and 80%:
  - < 50% wastes space
  - > 80% significant number of overflows

Dynamic Hashing

- How to grow and shrink hash tables?
- Alternative 1:
  - Use overflows and reorganizations
- Alternative 2:
  - Use dynamic hashing
  - Extensible Hash Tables
  - Linear Hash Tables

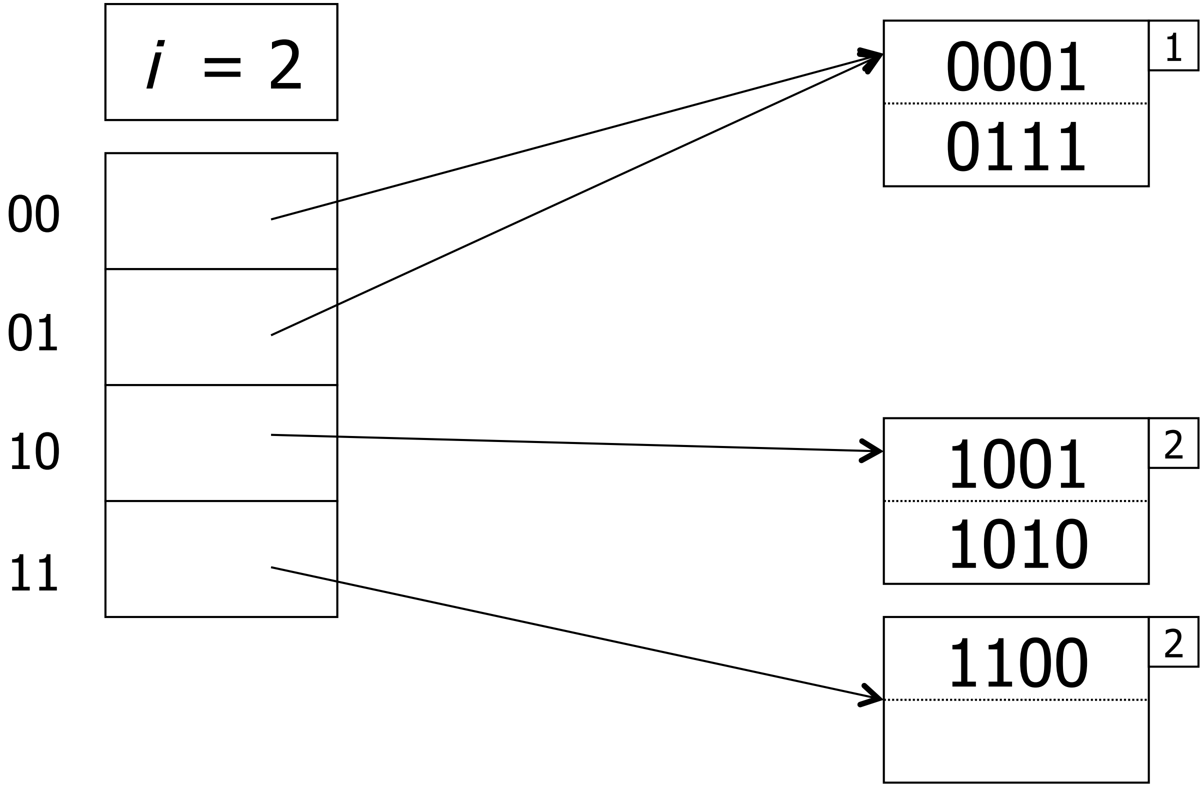
Extensible Hash Tables

- Hash function computes sequence of *k* bits for each key

$$k = 8 \quad \boxed{\underbrace{001}_{i\,=\,3}10101}$$

- At any time, use only the first *i* bits
- Introduce indirection by a pointer array
- Pointer array grows and shrinks (size 2^i)
- Pointers may share data blocks (store number of bits used for block in *j*)

Example: k = 4, f = 2



Insertion

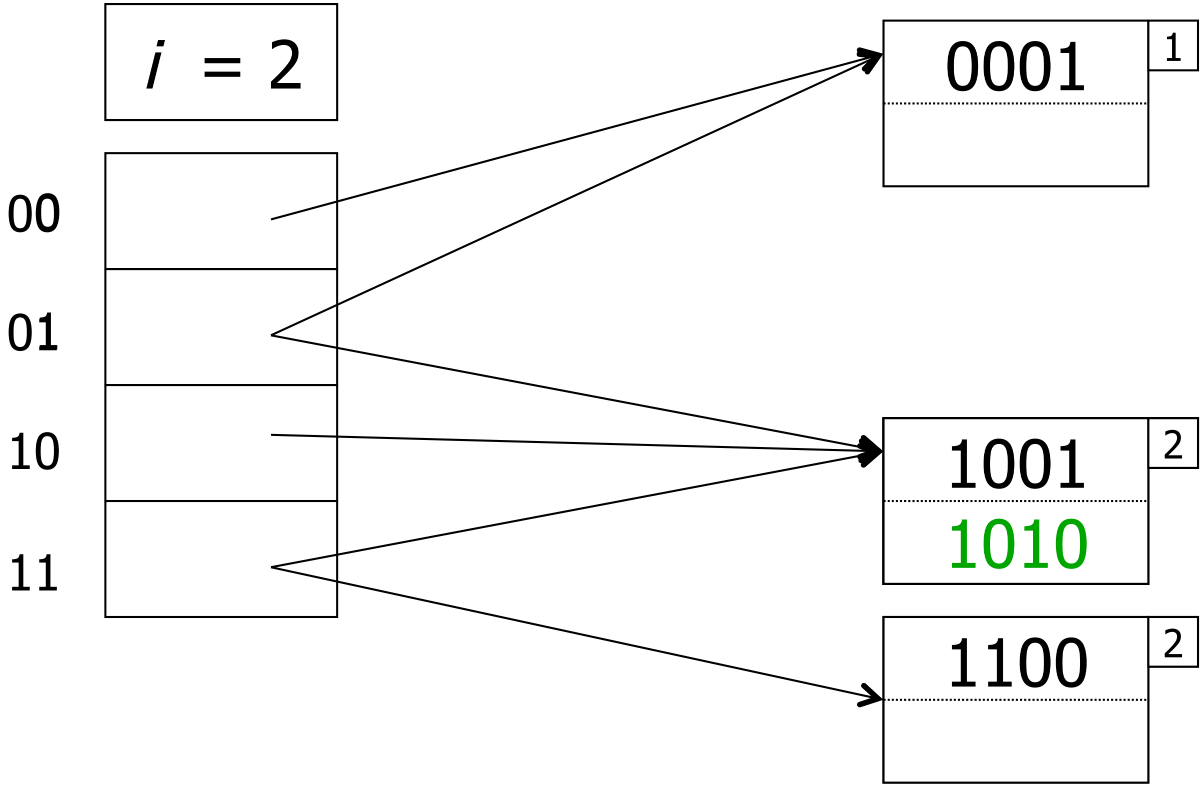
- Find destination block B for key-pointer pair
- If there is room, just insert it
- Otherwise, let *j* denote the number of bits used for block B
- If j = i, increment i by 1:
  - Double the length of the bucket array to 2^{i+1}
  - Adjust pointers such that for old bit strings w, w0 and w1 point to the same bucket
  - Retry insertion

Insertion

- If j < i, add a new block B‘:
  - Key-pointer pairs with (j+1)st bit = 0 stay in B
  - Key-pointer pairs with (j+1)st bit = 1 go to B‘
  - Set number of bits used to j+1 for B and B‘
  - Adjust pointers in bucket array such that if for all w where previously w0 and w1 pointed to B, now w1 points to B‘
  - Retry insertion

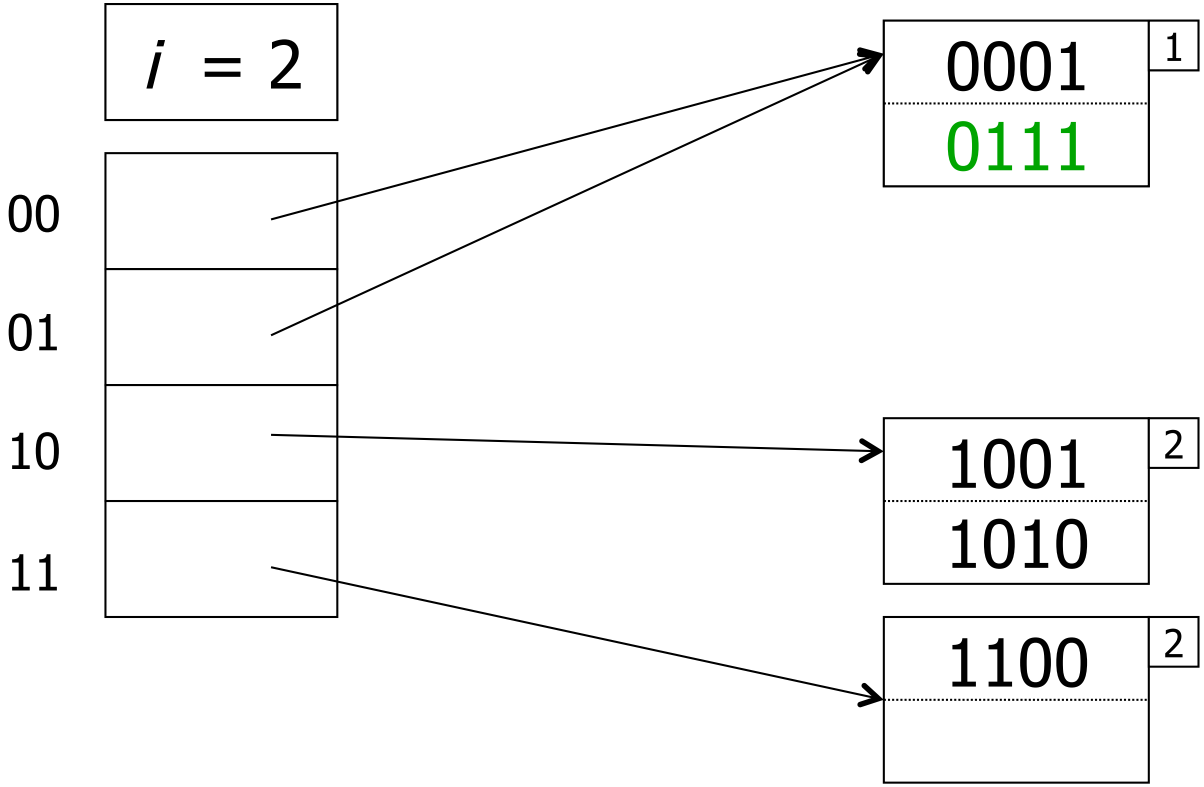
Example: Insert, k = 4, f = 2

- Insert 1010



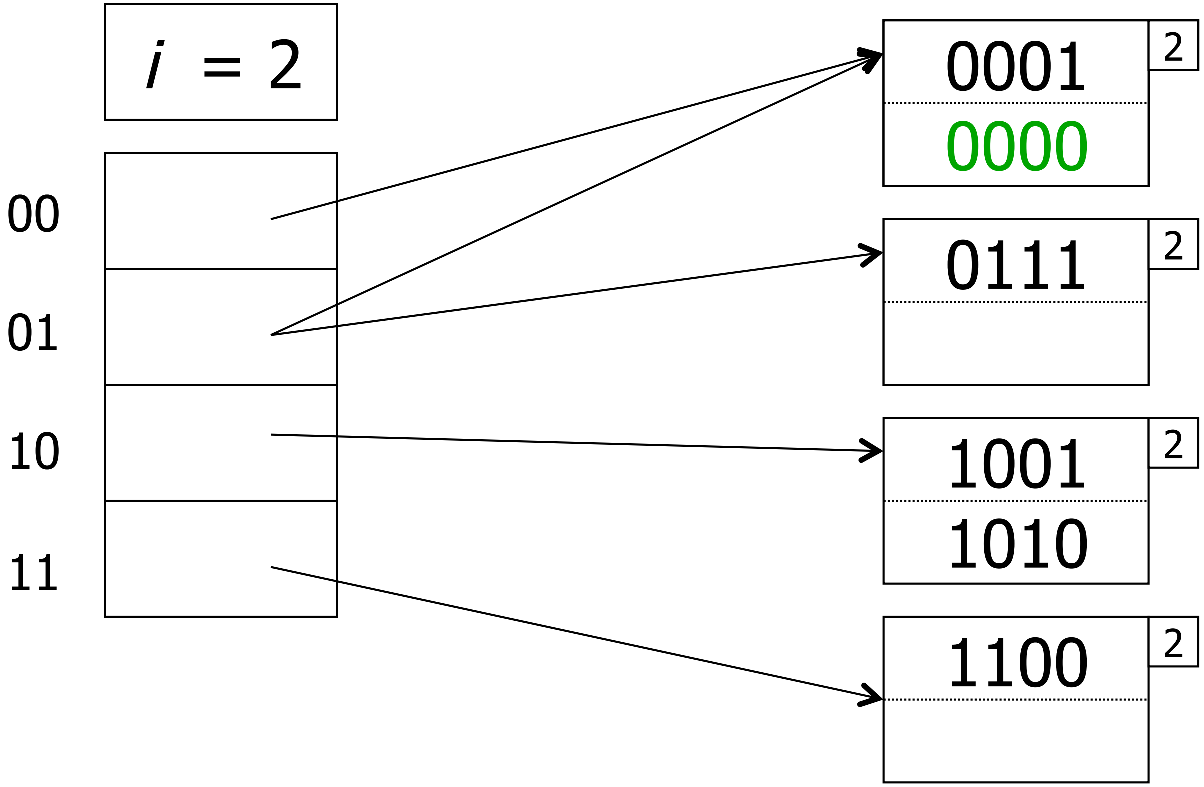
Example: Insert, k = 4, f = 2

- Insert 0111



Example: Insert, k = 4, f = 2

- Insert 0000

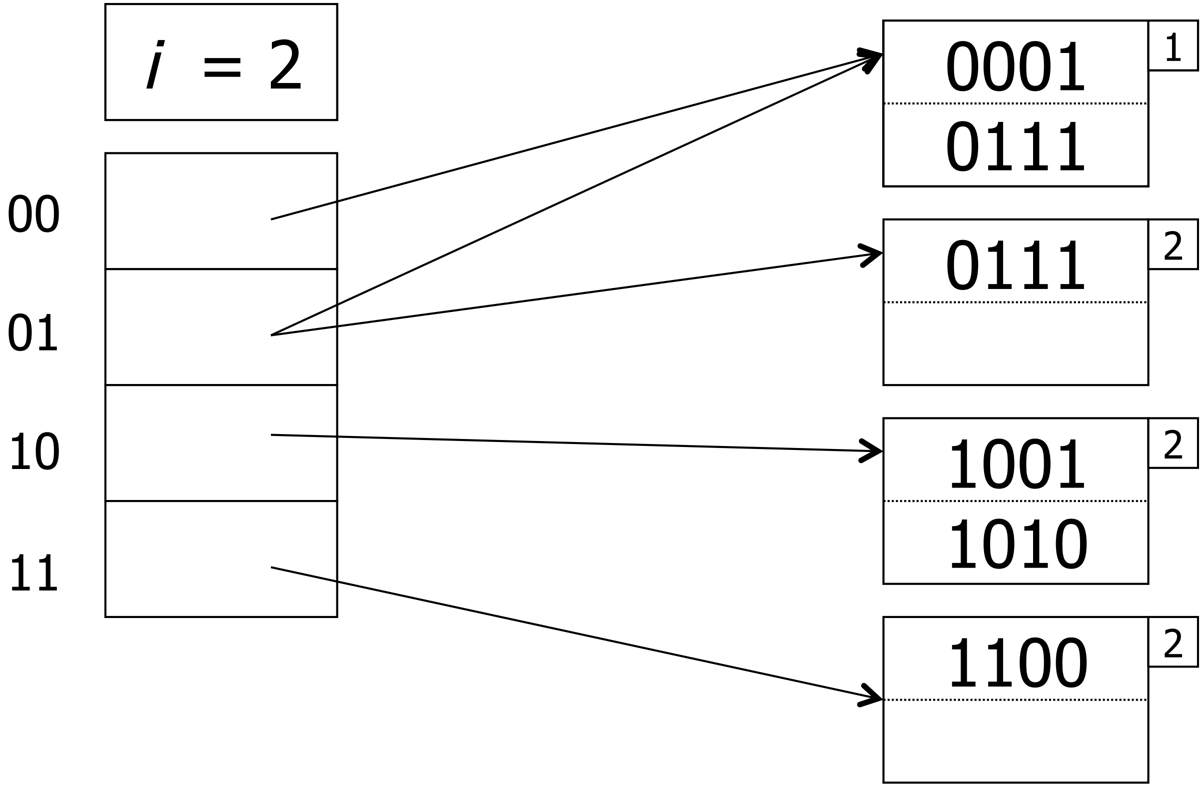


Deletion

- Find destination block B for key-pointer pair
- Delete the key-pointer pair
- If two blocks B referenced by w0 and w1 contain at most *f* keys, merge them, decrease their j by 1, and adjust pointers
- If there is no block with $j = i$, reduce the pointer array to size 2^{i-1} and decrease i by 1

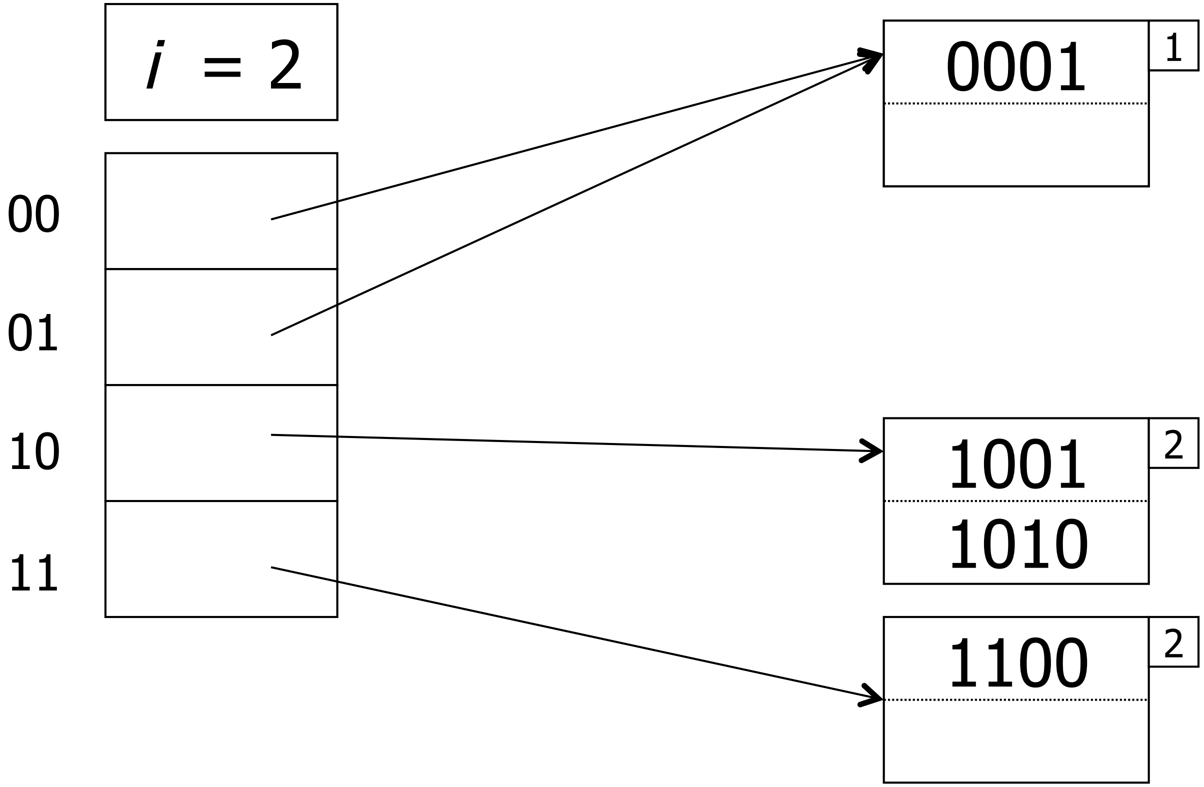
Example: Delete, k = 4, f = 2

- Delete 0000



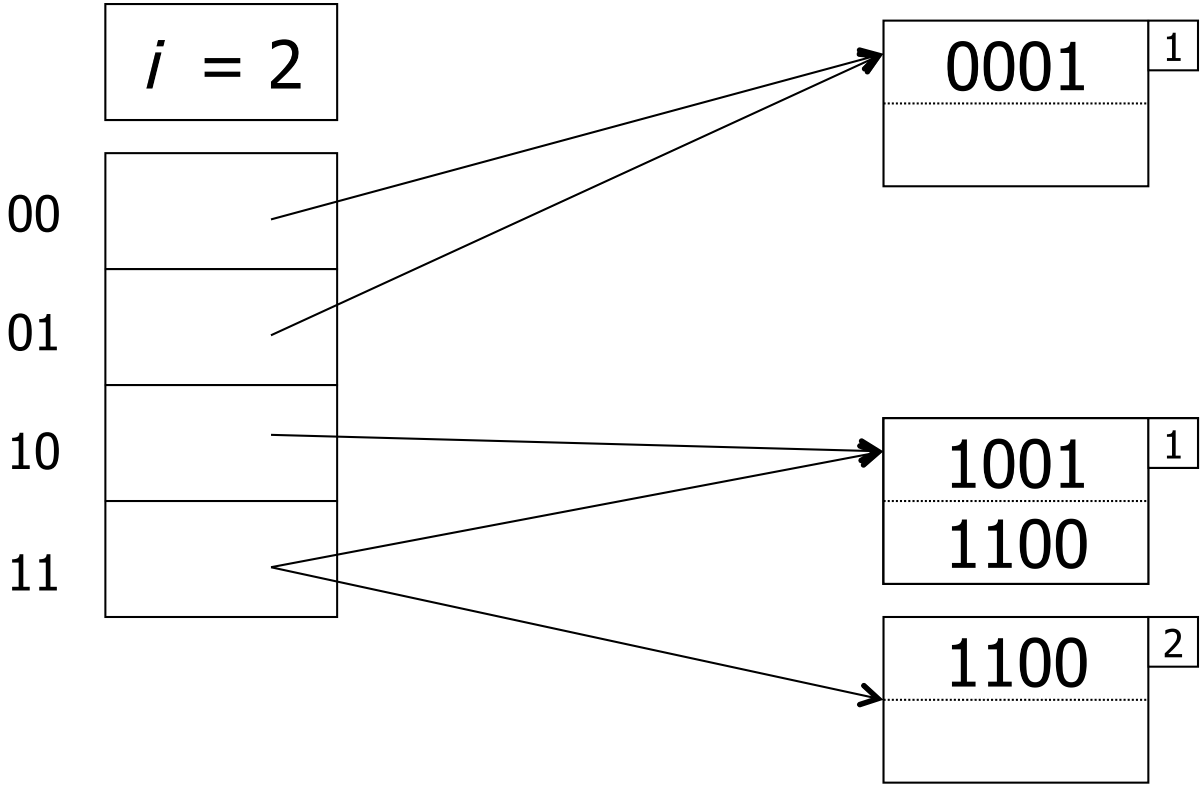
Example: Delete, k = 4, f = 2

- Delete 0111



Example: Delete, k = 4, f = 2

- Delete 1010



Efficiency

- As long as pointer array fits into memory and hash function behaves nicely, just need one I/O per lookup
- Overflows can still happen if many key-pointer pairs hash to the same bit string
- Solve by adding overflow blocks

Extensible Hash Tables

- Advantage:
  - Not too much waste of space
  - No full reorganizations needed
- Disadvantages:
  - Doubling the pointer array is expensive
  - Performance degrades abruptly (now it fits, next it does not)
  - For f = 2, k = 32, if there are 3 keys for which the first 20 bits agree, we already need a pointer array of size 1048576

Linear Hash Tables

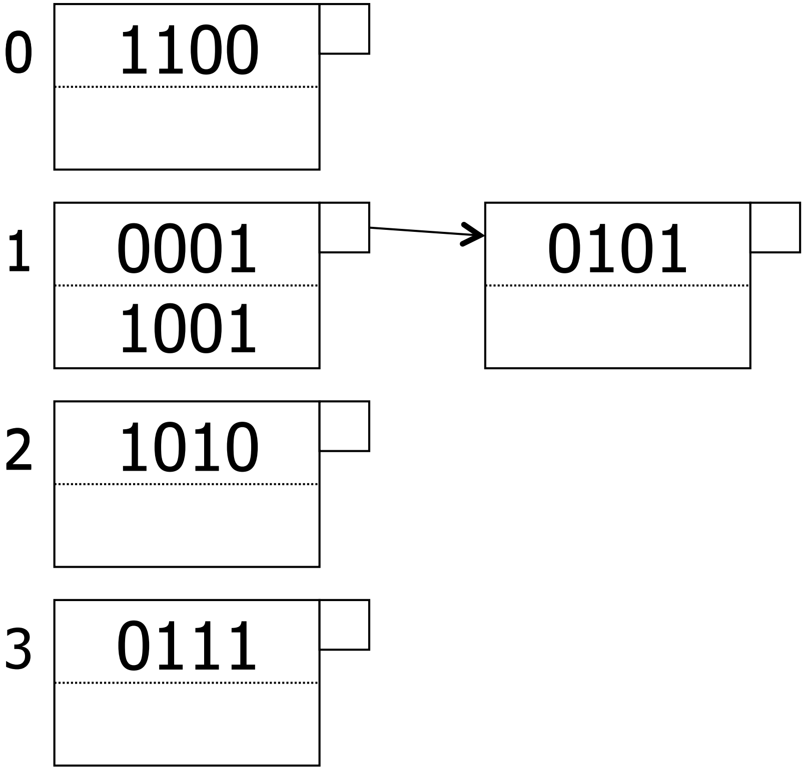
- Choose number of buckets *n* such that on average between for example 50% and 80% of a block contain records ($p_{min} = 0.5$, $p_{max} = 0.8$)
- Bookkeep number of records *r*
- Use ceiling($\log_2 n$) lower bits for addressing
- If the bit string used for addressing corresponds to integer *m* and $m \geq n$,
  use $m-2^{i-1}$ instead

Example: k = 4, f = 2

$i = 2$

$n = 4$

$r = 6$

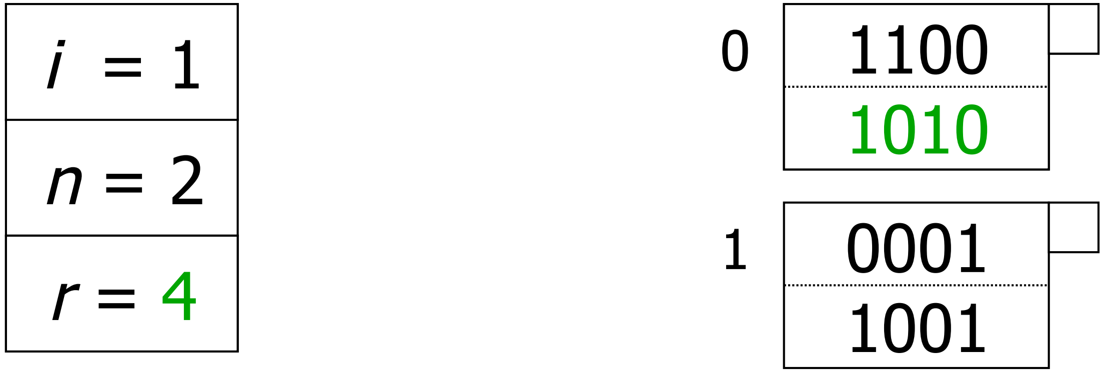


Insertion

- Find appropriate bucket (h(K) or $h(K)-2^{i-1}$)
- If there is room, insert the key-pointer pair
- Otherwise, create an overflow block and insert the key-pointer pair there
- Increase *r* by 1; if $r/n > p_{max}*f$, add bucket:
  - If the binary representation of *n* is $1a_2...a_i$, split bucket $0a_2...a_i$ according to the *i* -th bit
  - Increase *n* by 1
  - If $n > 2^i$, increase *i* by 1

Example: Insert, f = 2, p_{max} = 0.8

- Insert 1010



i = 1

n = 2

r = 4

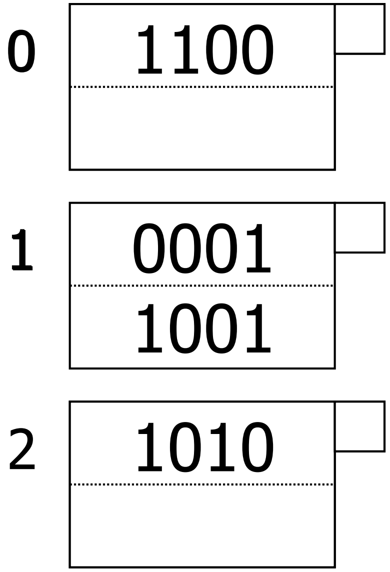
Example: Insert, f = 2, p_{max} = 0.8

- Attention: 4/2 > 1.6

i = 2

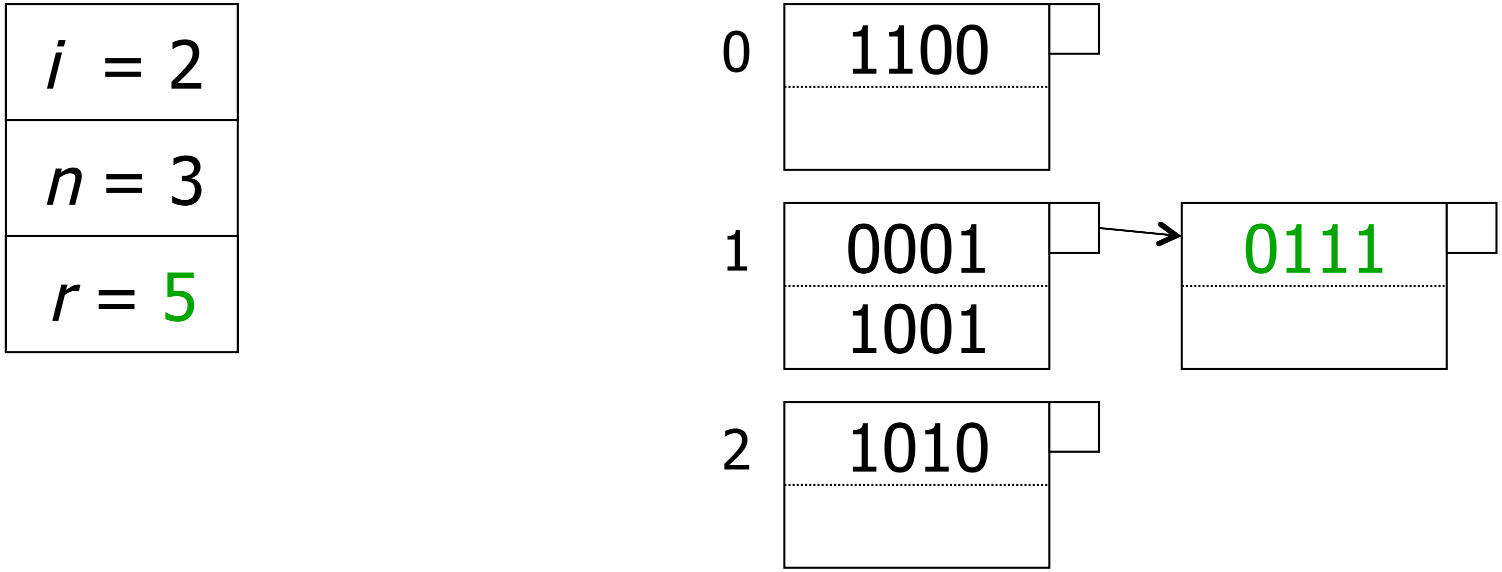
n = 3

r = 4



Example: Insert, f = 2, $p_{max} = 0.8$

- Insert 0111



$i = 2$
$n = 3$
$r = 5$

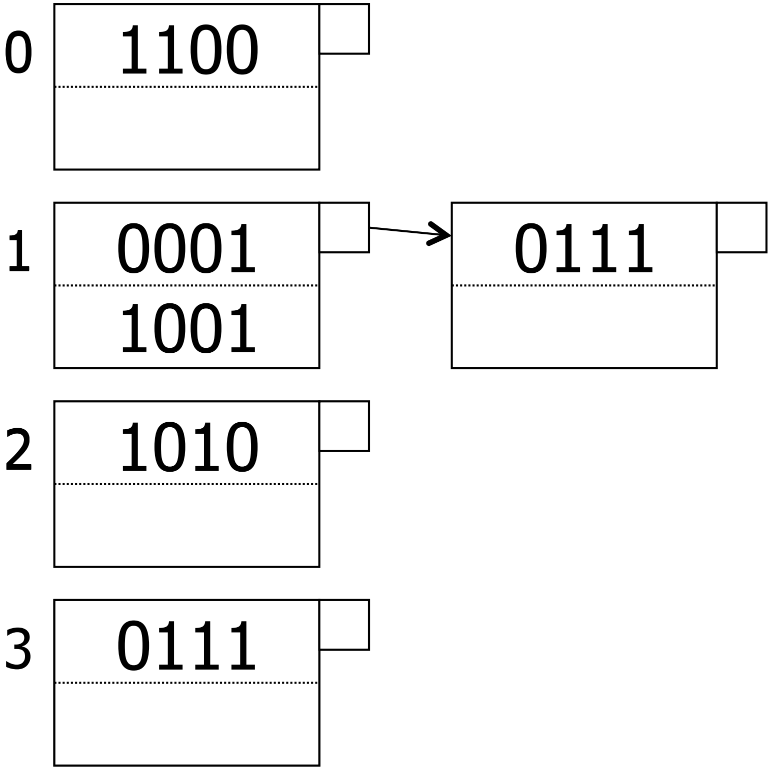
Example: Insert, f = 2, p_{max} = 0.8

- Attention: 5/3 > 1.6

i = 2

n = 4

r = 5



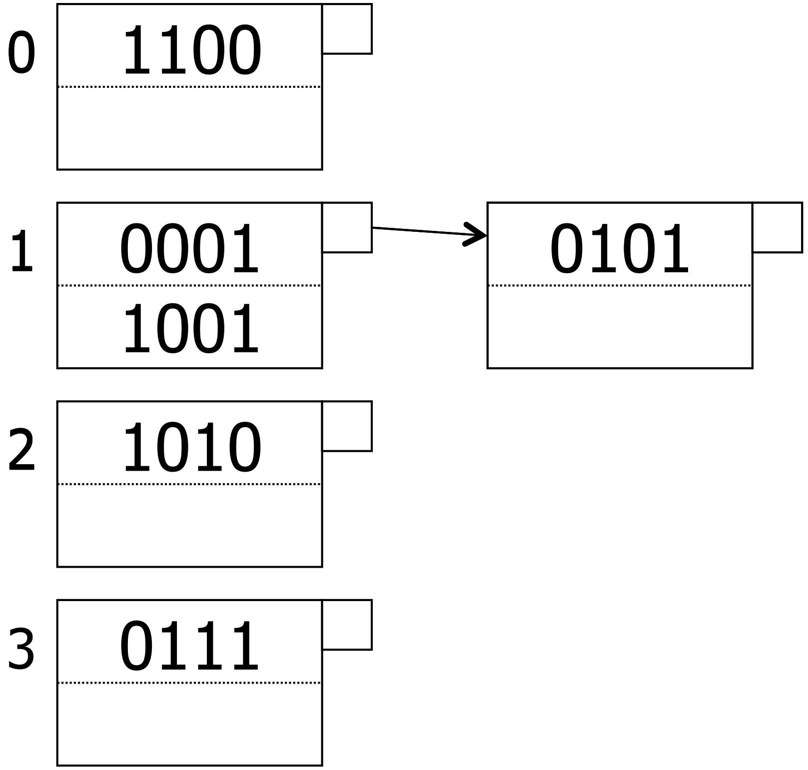
Example: Insert, f = 2, p_{max} = 0.8

- Insert 0101

i = 2

n = 4

r = 6



0 1100

1 0001 1001 → 0101

2 1010

3 0111

Linear Hash Tables

- Advantage:
  - Not too much waste of space
  - No full reorganizations needed
  - No indirections needed
- Disadvantages:
  - Can still have overflow chains

B+Trees vs Hashing

- Hashing good for given key values
- Example:
  SELECT * FROM Sells WHERE price = 20;
- B+Trees and conventional indexes good for range queries:
- Example:
  SELECT * FROM Sells WHERE price > 20;

Summary 11

More things you should know:

- Hashing in Secondary Storage
- Extensible Hashing
- Linear Hashing

THE END

Important upcoming events

- March 20: delivery of the final report
- March 26: 24-hour take-home exam
- March 28: Øltesten!