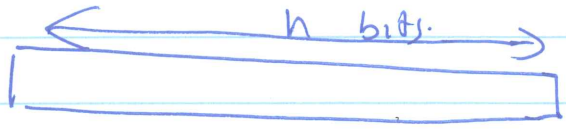
Dynamic Hashing

4

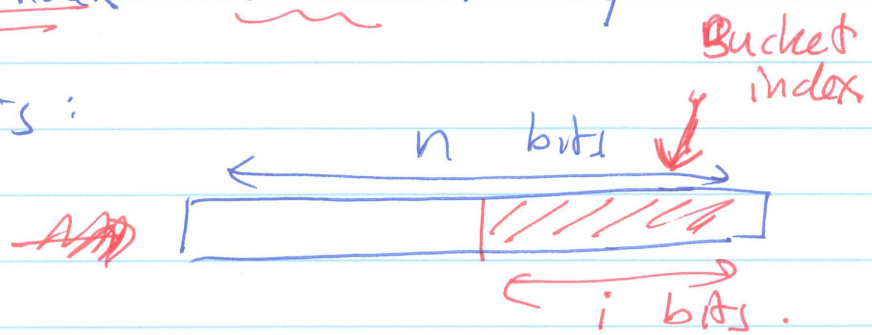
Extensible Hash Table

- Th hash function

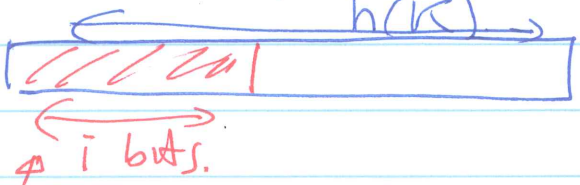
$h(K)$ = number of n bits.

= 

- The bucket index determined by $h(K)$ is:



(NB: you can use the upper i-bit also.



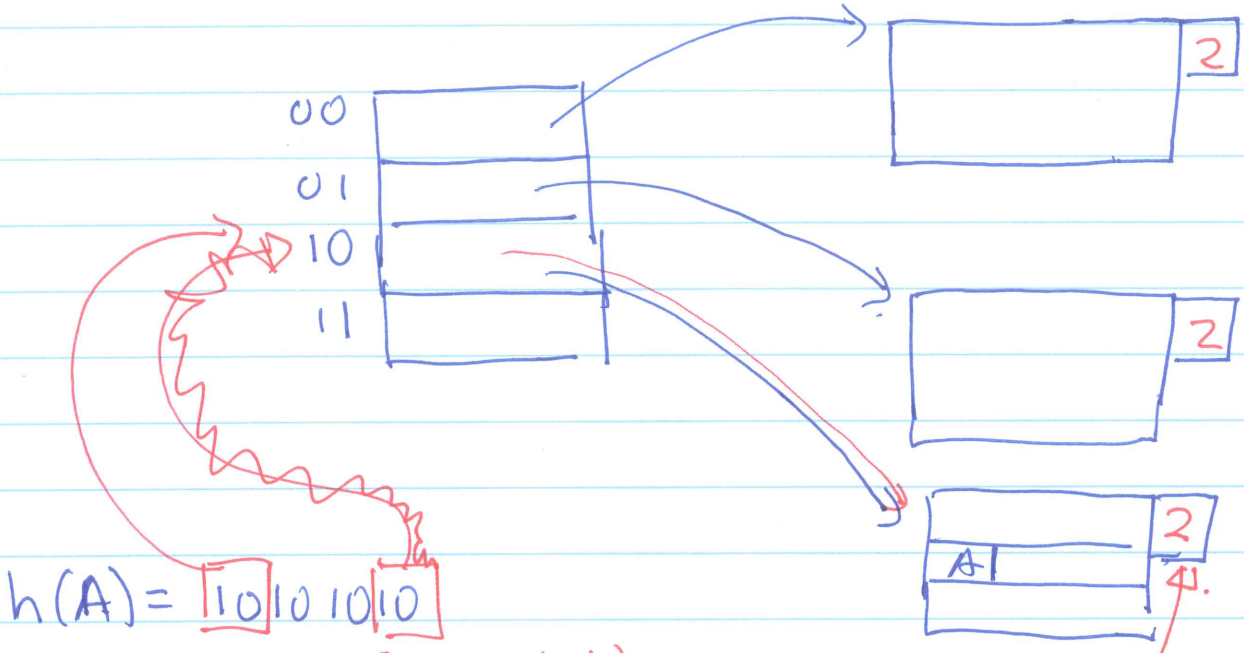
)

- New: Each data block contains a number indicating

[# bits of hash function]

used (to find record in data block)

Example: suppose i = 2 (2 bits used).



look use the first i bits

Keep examples that way.

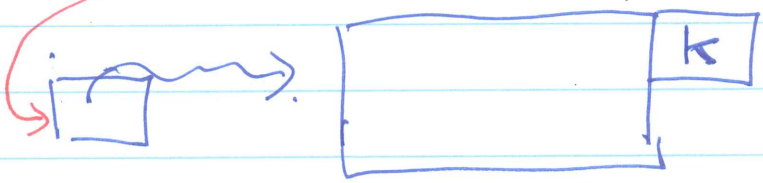
Inserting in Extensible Hash Table:

insert(x).

(1) Compute hash function $h(x)$

(2) Use i-bits of $h(x)$ = 

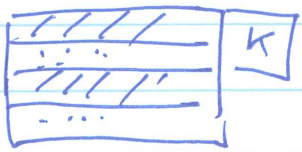
to find data block.

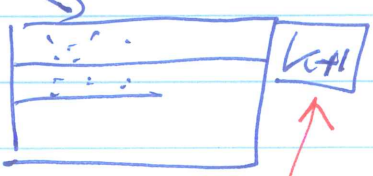


(3) If data block has space for record
→ insert + Done.

(4) If data block is full.

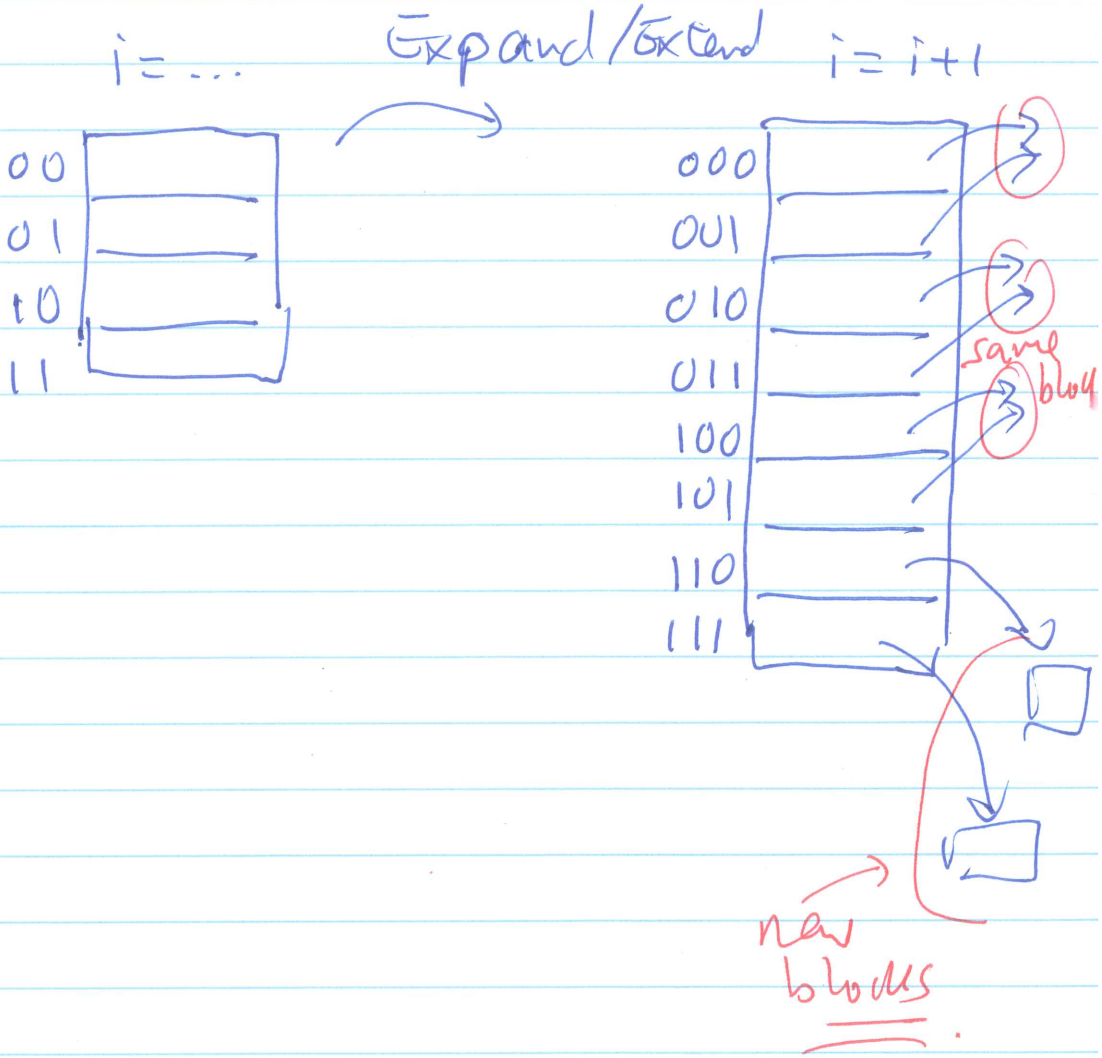
(A) split block using k+1 bits of $h(K)$.





(B) tag new block with (k+1)

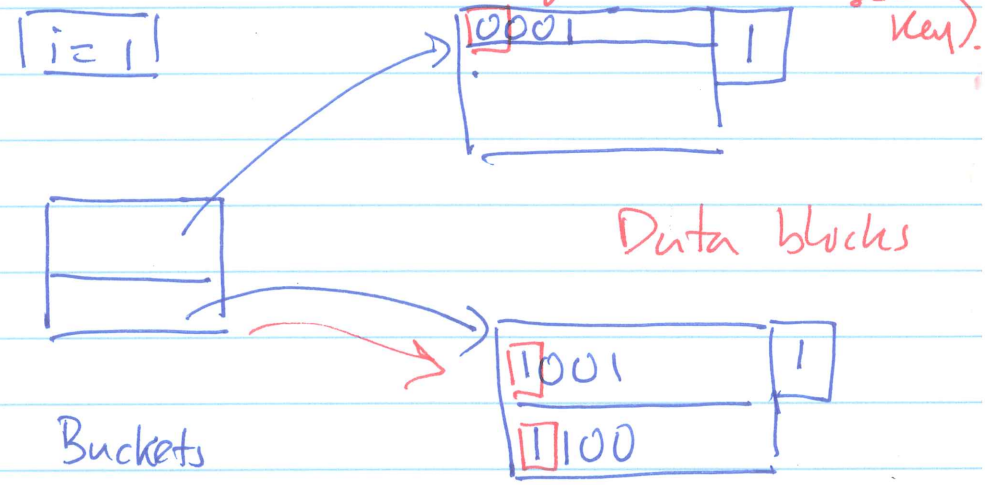
(c) If $(k+1) > 1$ then


i = ...
Expand/Extend
i = i+1
00
01
10
11
000
001
010
011
100
101
110
111
same block
new blocks

Example: Insert

Originally: 3 data records hashed.
i = 1.

(1) State:



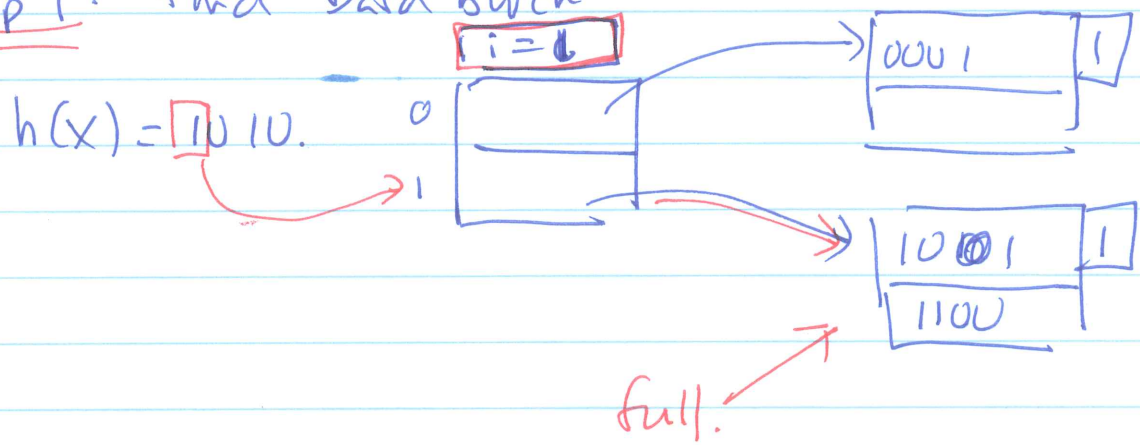
(2) Insert a record ~~#~~ X, with:

h(X) = 1010

cause overflow
↓
slip overflow block.

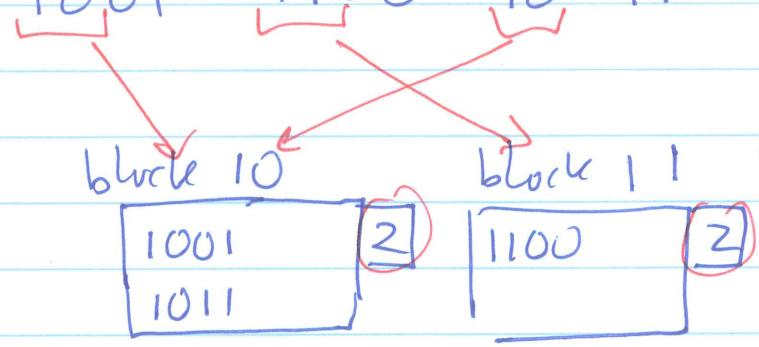
Split overflow block + Expand bucket away

Step 1: Find Data block:



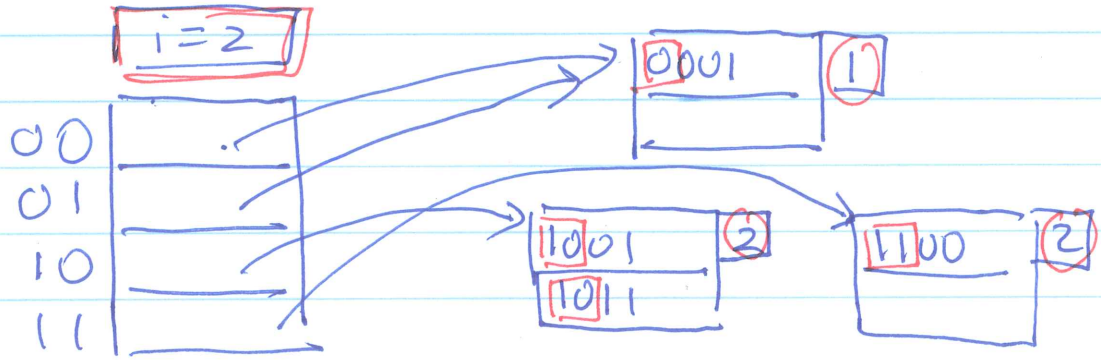
Step 2: split data block on $h(K)$ based on one more bit.

keys: 1001 1100 1011.

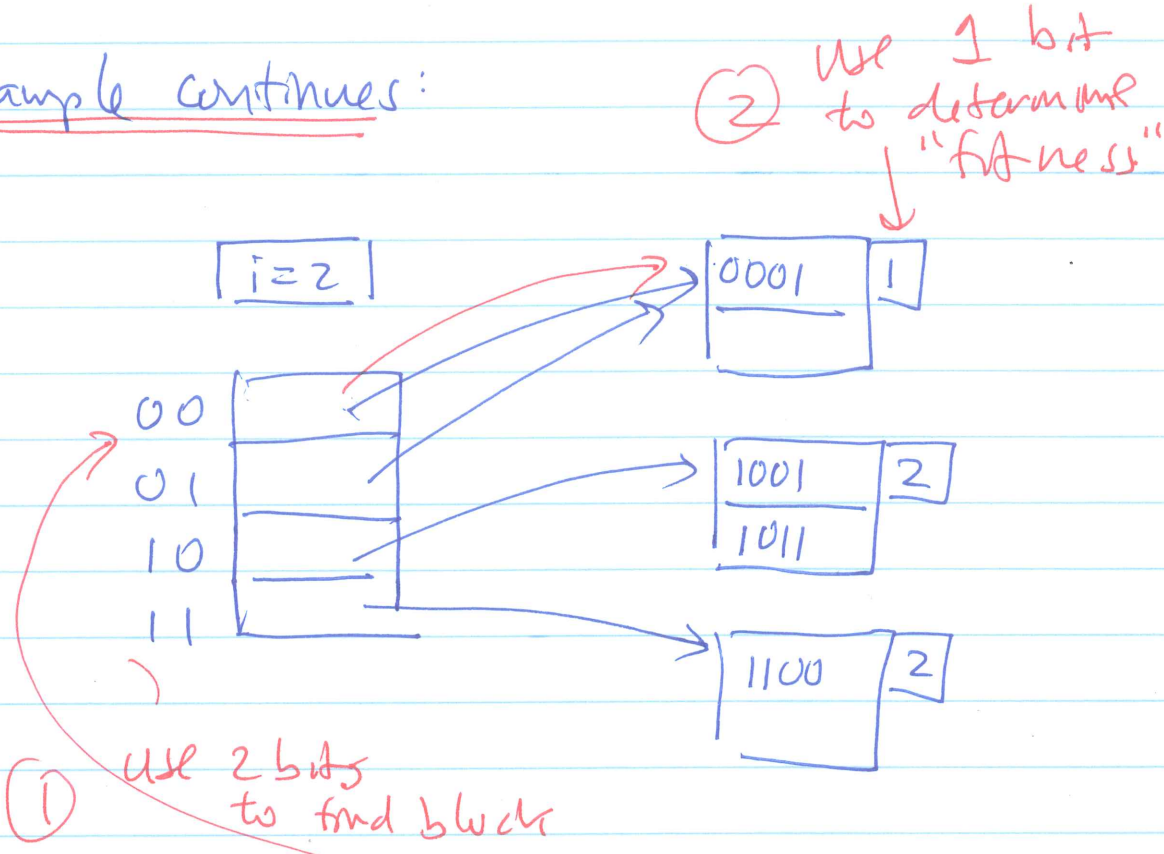


2 > 1

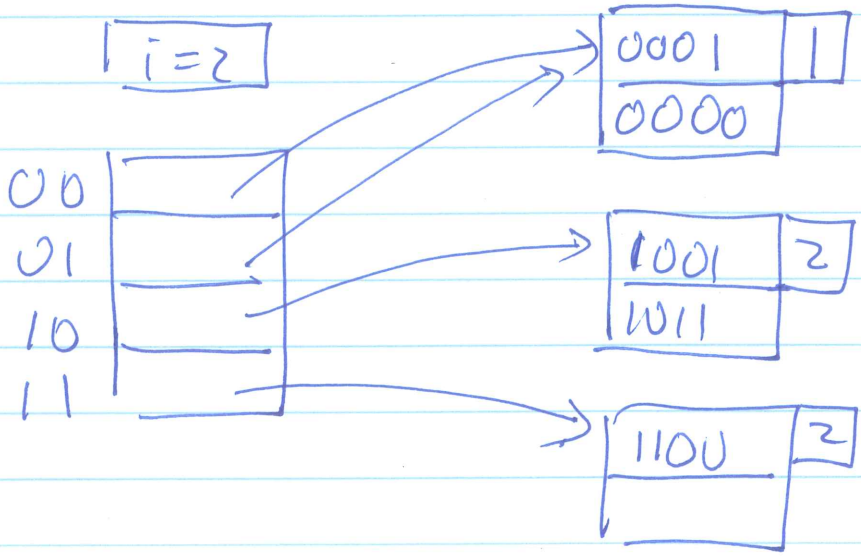
Step 3: ~~double~~ Expand bucket away with extra bit + update block pointers.



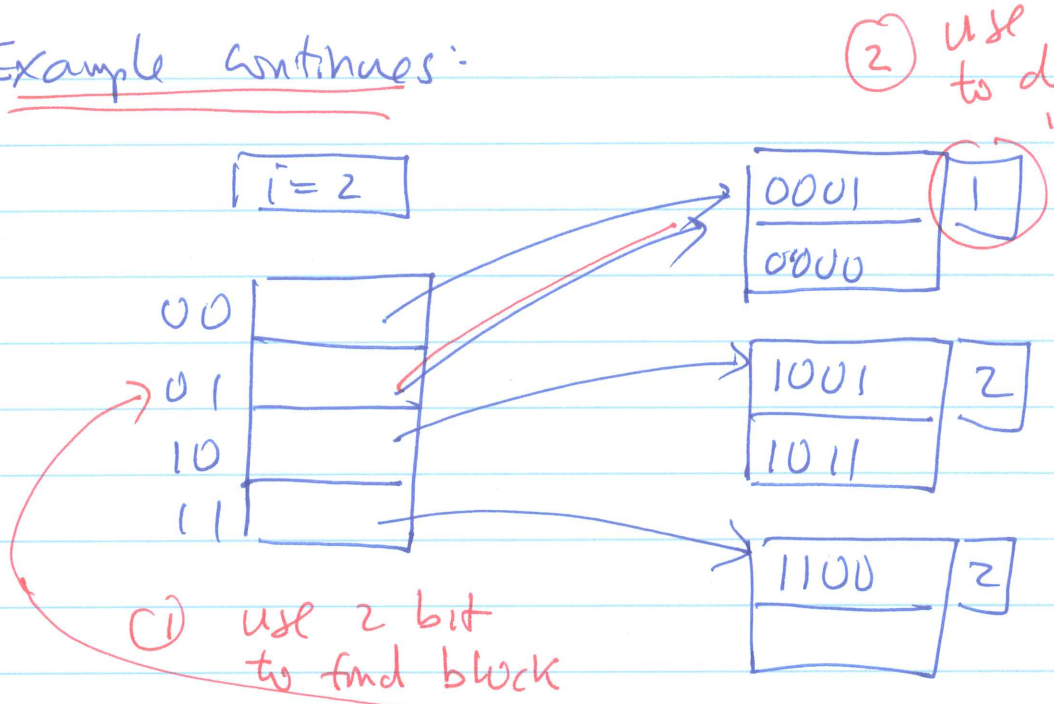
Example continues:



- Insert (A), such that h(A) = 10000
  i=2
- Result:



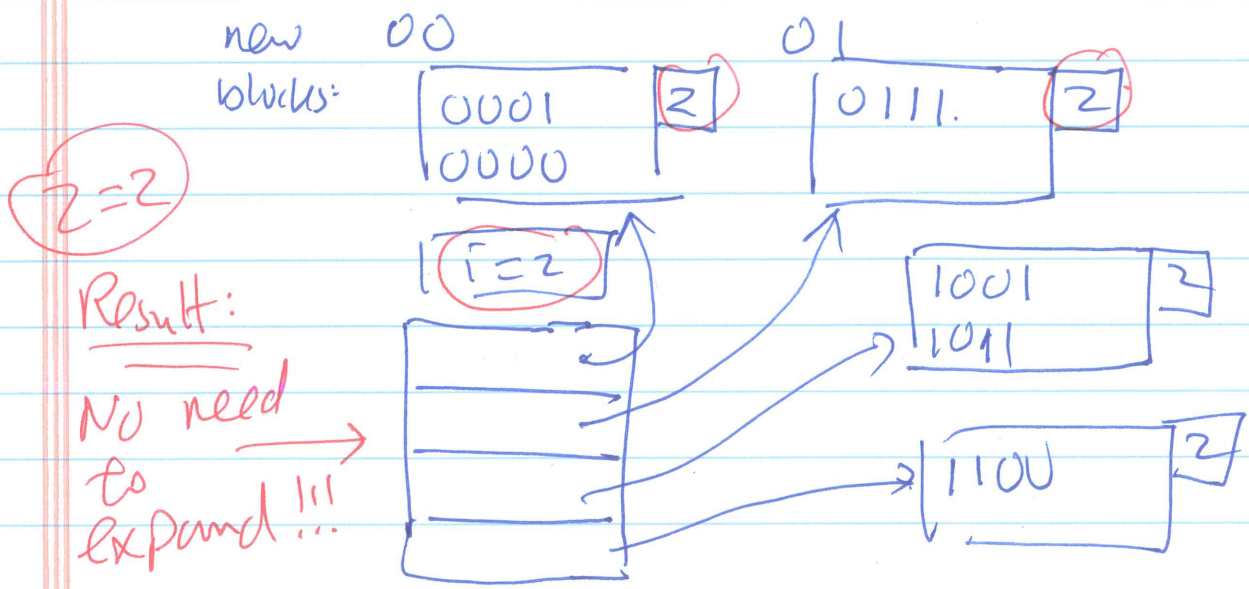
Example continues:



- Insert(B), such that: $h(B) = \boxed{01}11$ (i=2)

- cause overflow
  ↓
- Split block using [11] +1 bit ⇒ 2.

- keys: 0001 0000 0111



Linear Hash Tables

- Extensible hash table's problem:
  - ☆ "exponential growth"
    ↓
    the # buckets doubles
  - ☆ Large overhead (pointer copying) when ~~tab~~ buckets doubles.

(Advantage: never need more than 1 data block read to access a record.

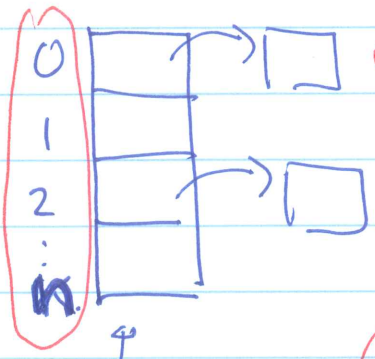
(assuming bucket table can be stored in memory).)

- Slower growth rate:

  linear hash tables.

Characteristics of Linear Hashing:

(1)



Use overflow blocks

n buckets

$k = \log(n)$

bits to number n. (Duh???).

Avg occupancy in bucket. (# records = fixed)

buckets (n) varies so that:

avg. occupancy per bucket = fixed (eg 80%).

(2) Full blocks cannot always be split.

So: Linear Hashing makes use of overflow blocks.

(3) # bits used to number the bucket away (index)

$$= \lceil \log_2(n) \rceil$$

n = current # buckets.

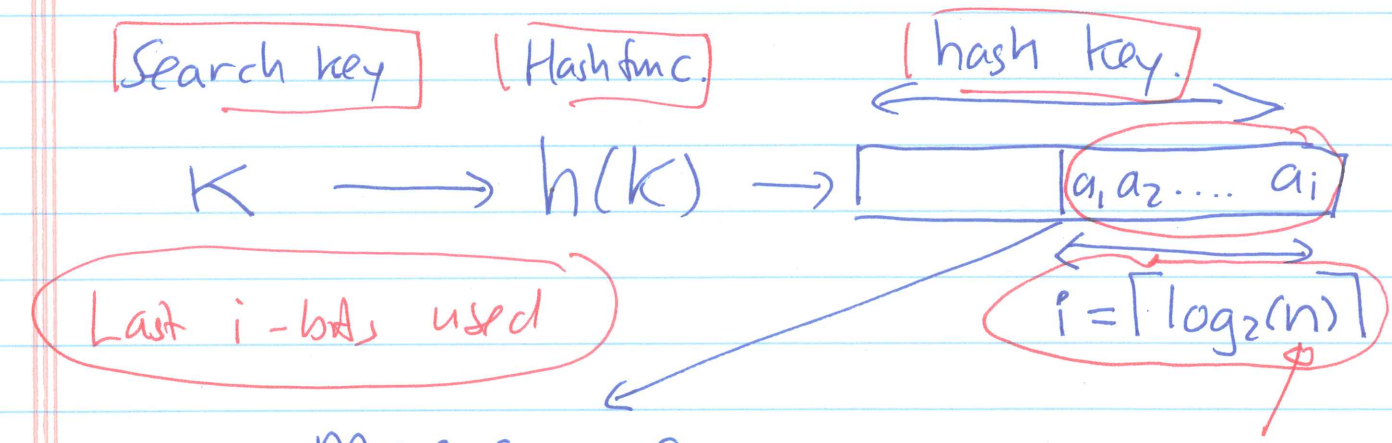
important point →

bits in hash key used:

$$i = \lceil \log_2(n) \rceil$$

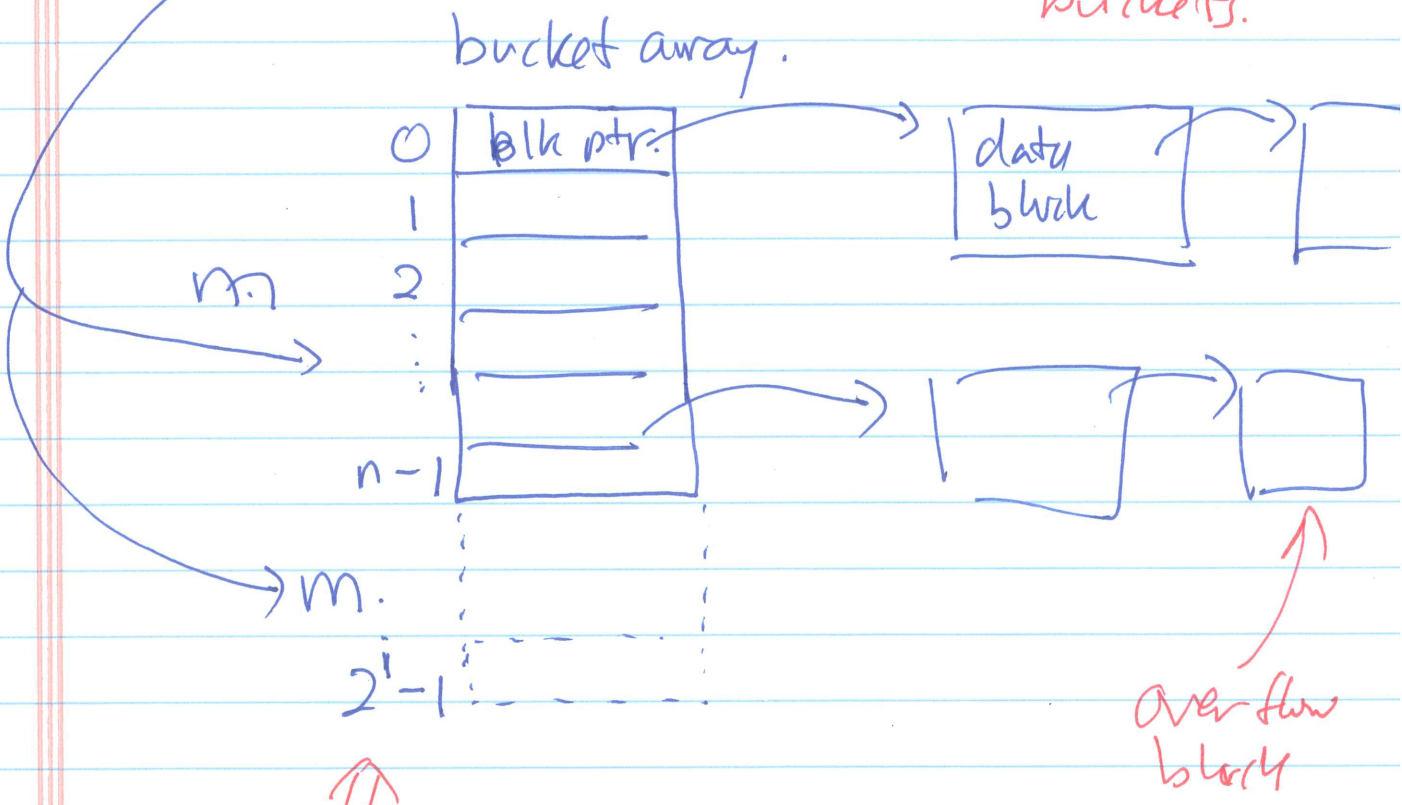
curr. # buckets

(4) Important: (overview).



$m = a_1 a_2 \ldots a_i$

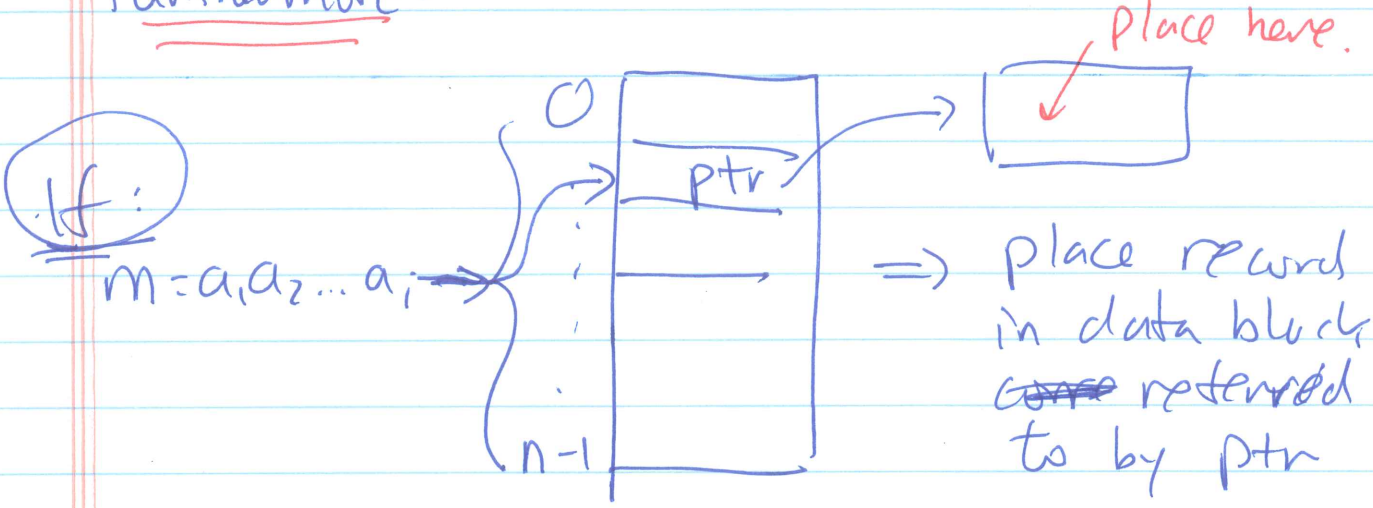
n = current # of buckets.

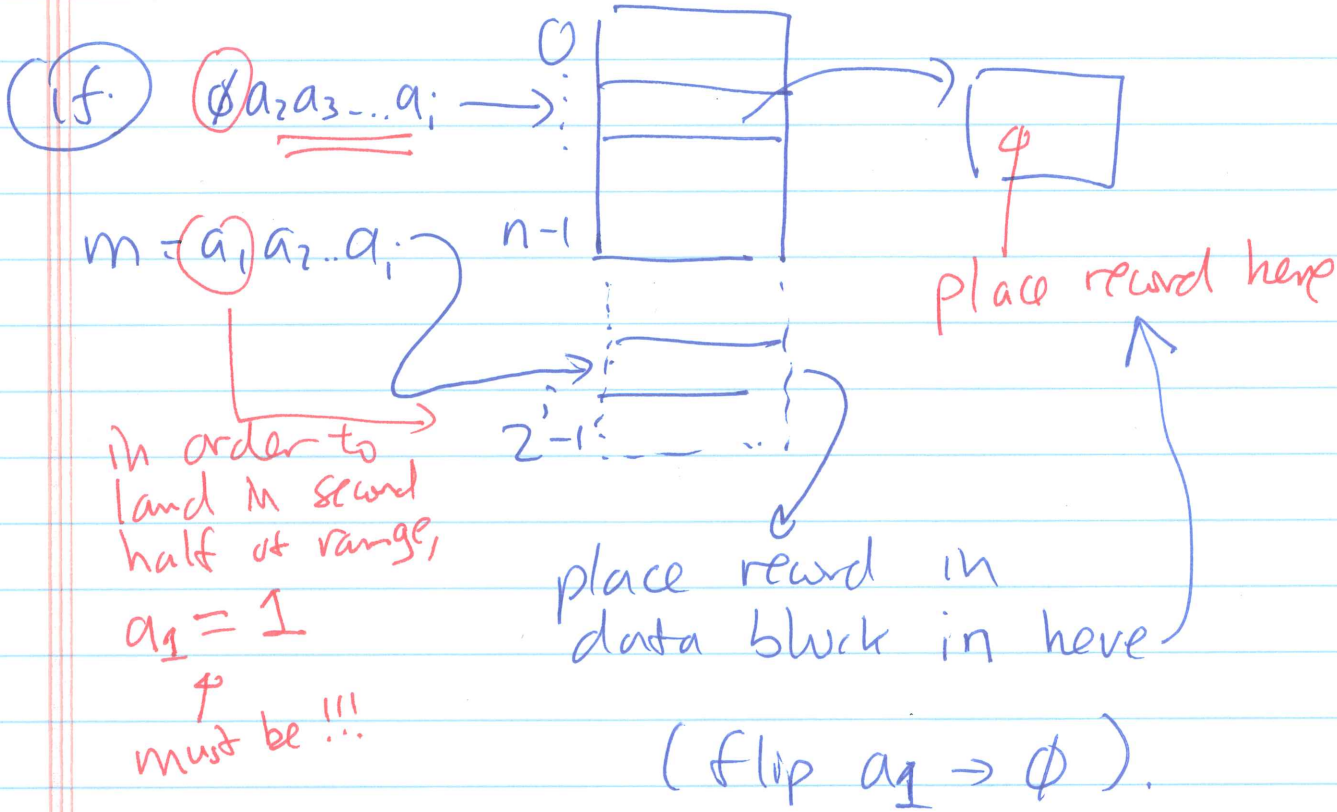


Note: $m = a_1 a_2 \ldots a_i$ CAN be larger than $n-1$!!!

n = current # buckets

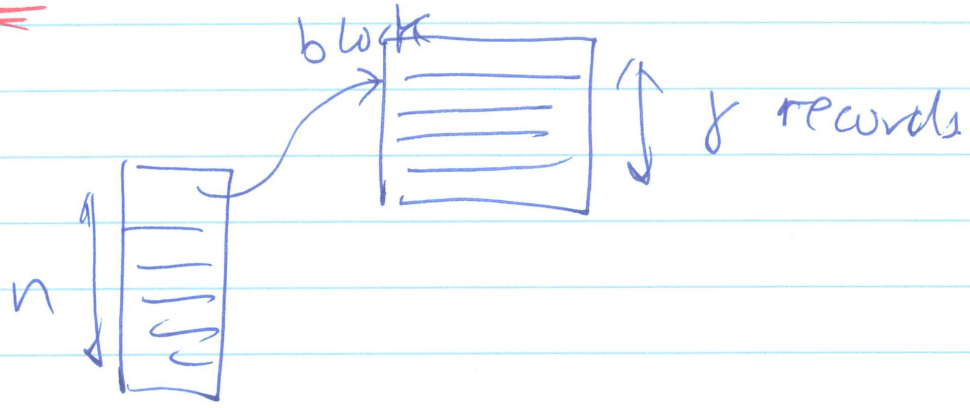
Furthermore:





How to compute avg. occupancy ratio

Let f = # records in a block.



n = current # buckets.

Capacity of all buckets = $n \cdot f$.

Let r = current total # records.

avg. occupancy ratio:

$$\alpha = \frac{r}{n \cdot f} \leq C$$

C: chosen threshold

Example: (Linear Hashing.)

Suppose we choose:

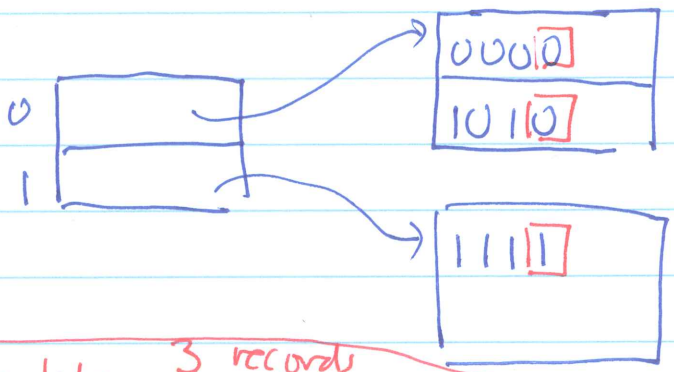
Avg. occupancy per bucket = 0.85 (85%)

Given: (2) records in 1 bucket.

⇒ avg # records in 1 bucket = 1.7

Example state: (r = 3 records).

i = 1
n = 2
r = 3



Avg. occ per bucket = $\frac{3 \text{ records}}{4 \text{ max}} = 0.75 \leq 0.85$.

parameters:

i = # bits in hash key used to determine bucket

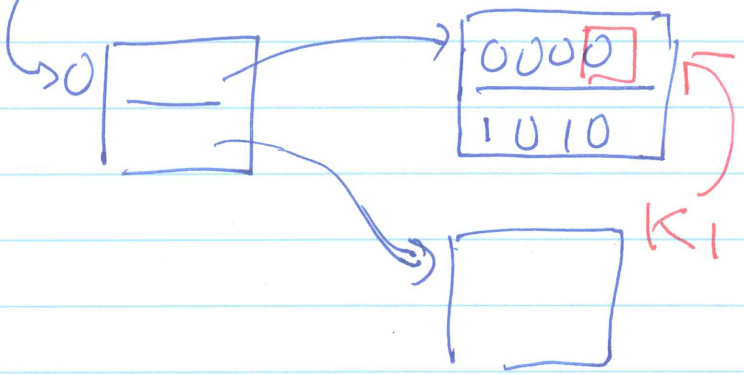
n = current # of buckets

r = current # of records.

Explanation:

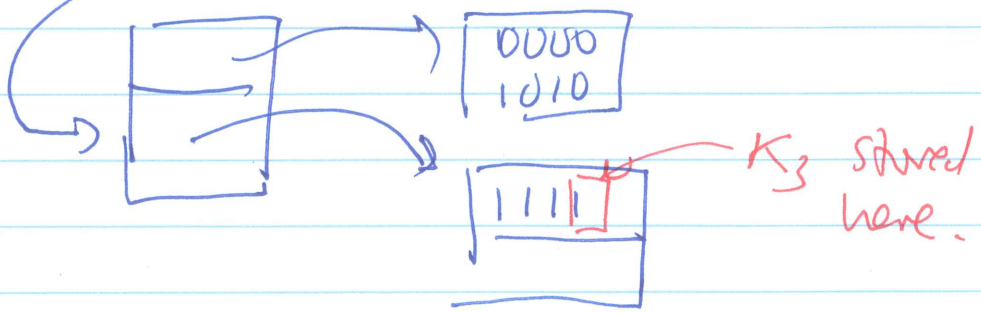
K_1 : $h(K_1) = 0000$

use $i=1$ bit to determine bucket



K_3 : $h(K_3) = 1111$

use $i=1$ bit to determine bucket.



Inserting into Linear Hash Tables.

Current state:

predetermined.

$i = i$

$n = n$

$r = r$ (# records)

γ = # rec's in 1 block.

$$\frac{r}{n\gamma} \leq \tau$$

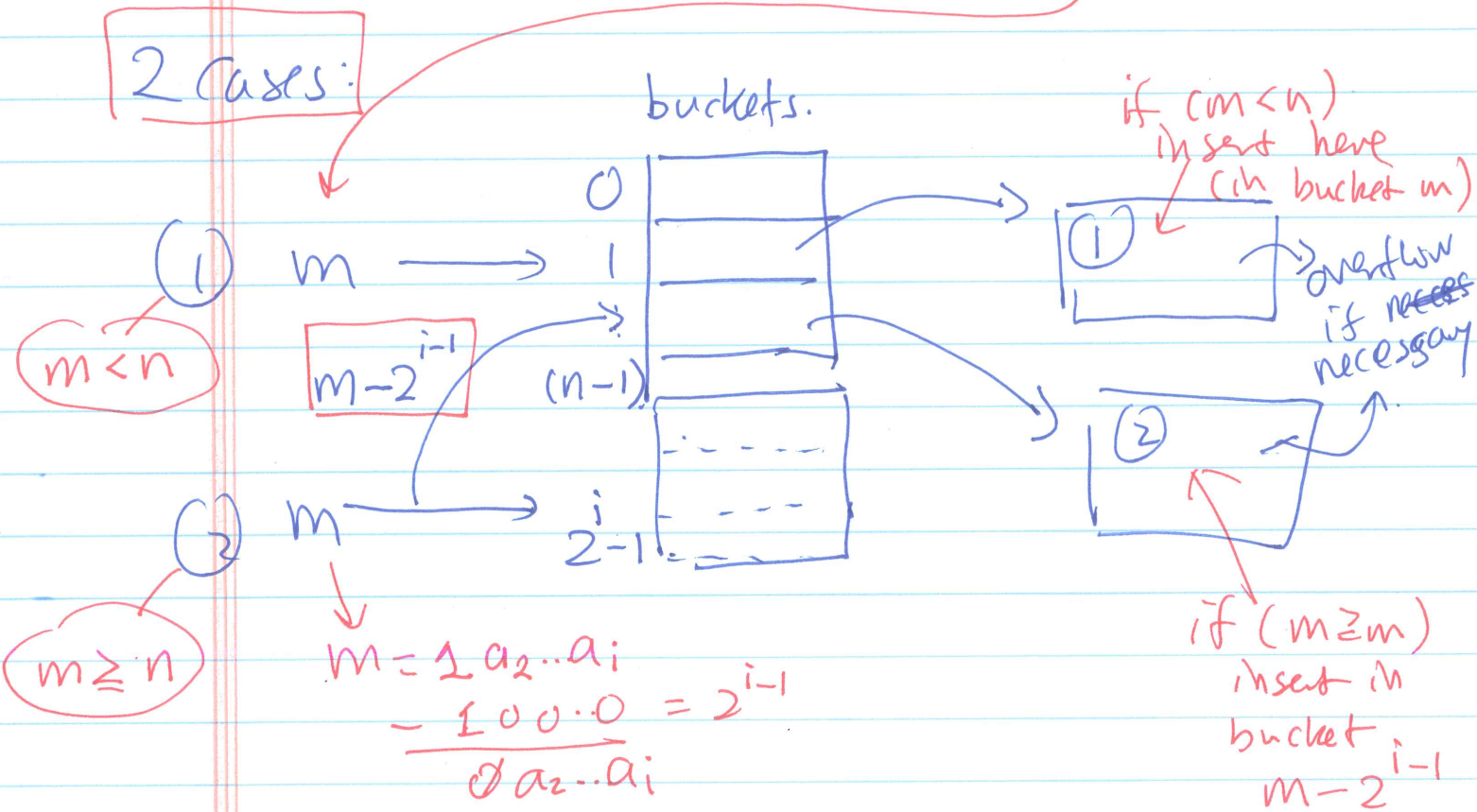
Insert Record with search key K:

hash key.

$K \rightarrow h(K) = \boxed{\cdots\cdots \; a_1 a_2 \cdots a_i}$

$m = a_1 a_2 \cdots a_i$

2 cases:



After insertion, test:

$r' = r + 1$.

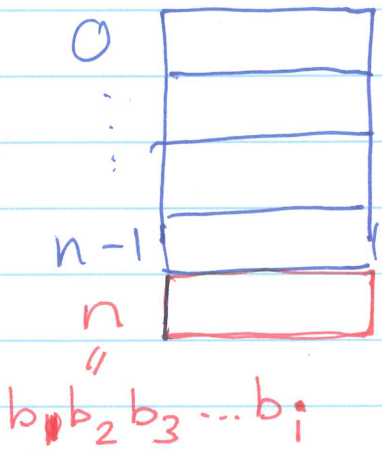
$$\frac{r'}{n\gamma} \leq \tau.$$

If:

$$\frac{r'}{n\gamma} > \tau$$

Then:

(1) Increase buckets by 1: $(n' = n+1)$



Note: this bucket has NOTHING to do with bucket $a_1 a_2 \ldots a_i$ (used in last insertion!)

(2) If $b_1 = 1$ then:

{

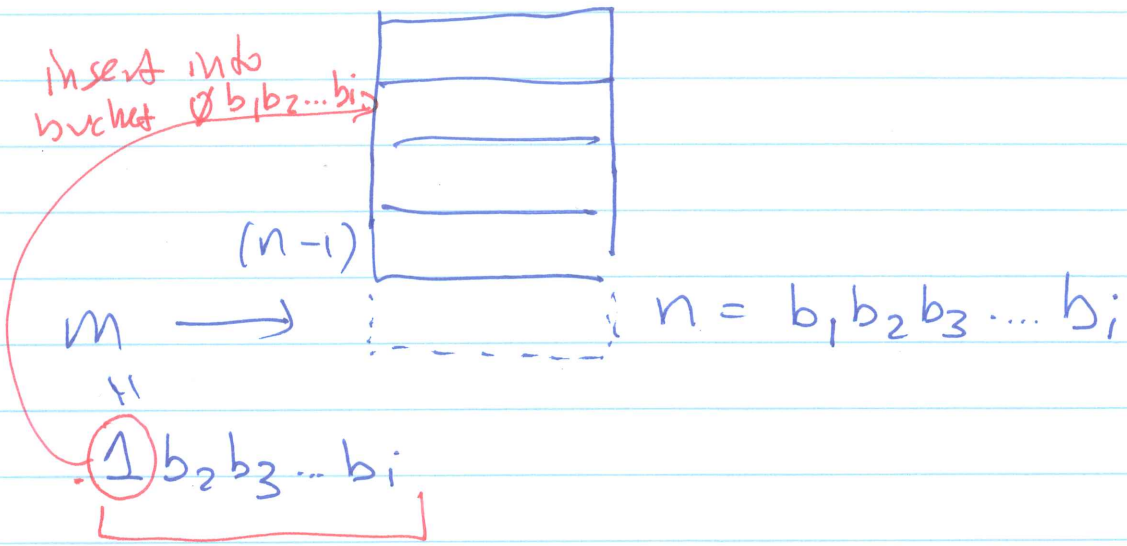
for every record in bucket $\emptyset b_2 b_3 \ldots b_i$

do: move records with key $1 b_2 b_3 \ldots b_i$ to new bucket.

}

Reason for step (2)

(A) the insertion step does this:
(Before adding bucket n+1)



(B) Now that we have added a real bucket $n = \underbrace{b_1}_{1} b_2 b_3 \ldots b_i$

we must MOVE the

"temporarily housed" tuples

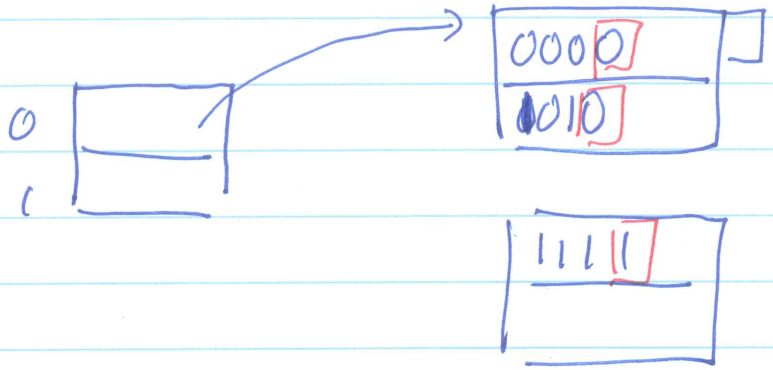
into their "real" bucket !!!

Example: parameters: $i = 1$ $\tau = 0.85$

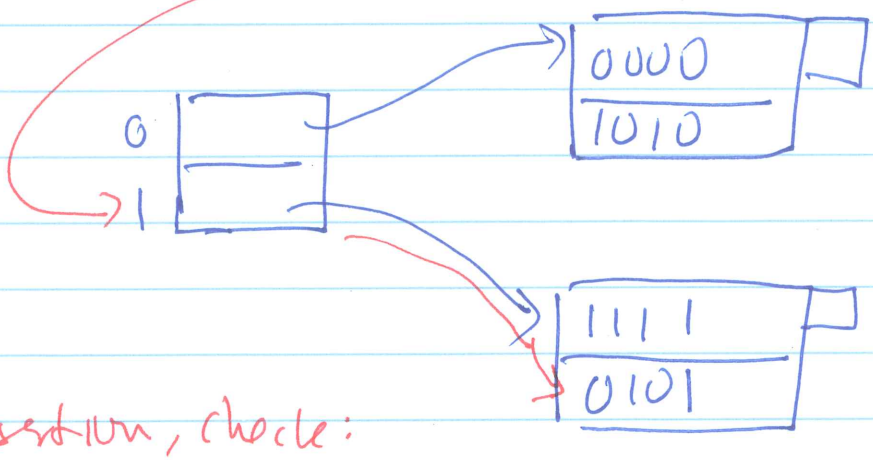
$n = 2$

$r = 3$ (2 records per block)

Currently:



Insert K, such that $h(K) = 0101$.

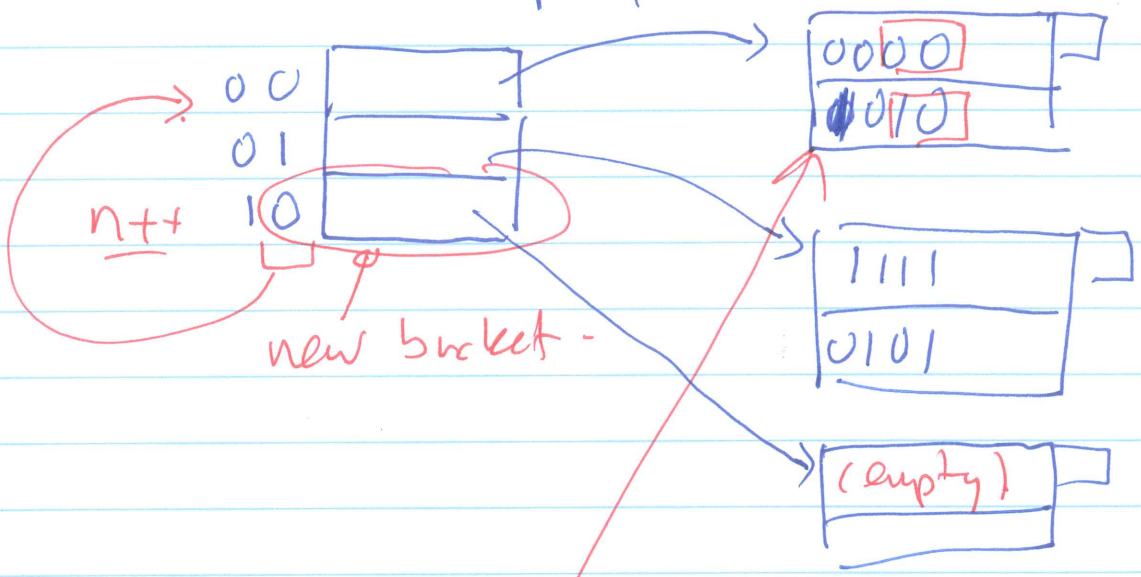


After insertion, check:

$$\frac{r}{n_y} = \frac{4}{2 \cdot 2} = \frac{4}{4} = 1 > 0.85$$

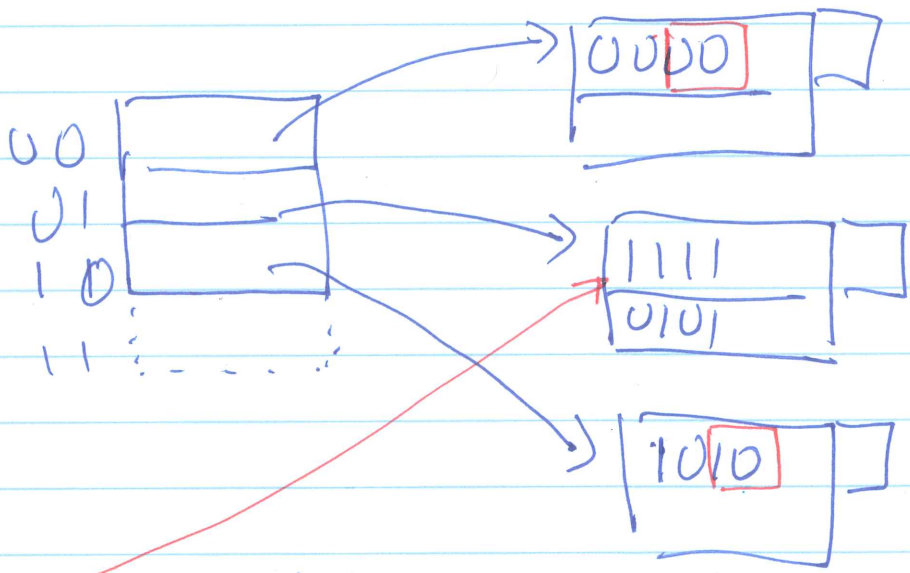
$\Rightarrow$ add 1 more bucket

New parameters: $i=2$ $n=3$ $r=4$. $\tau=0.8$



Split this bucket on key based on 2 digits.

Result:

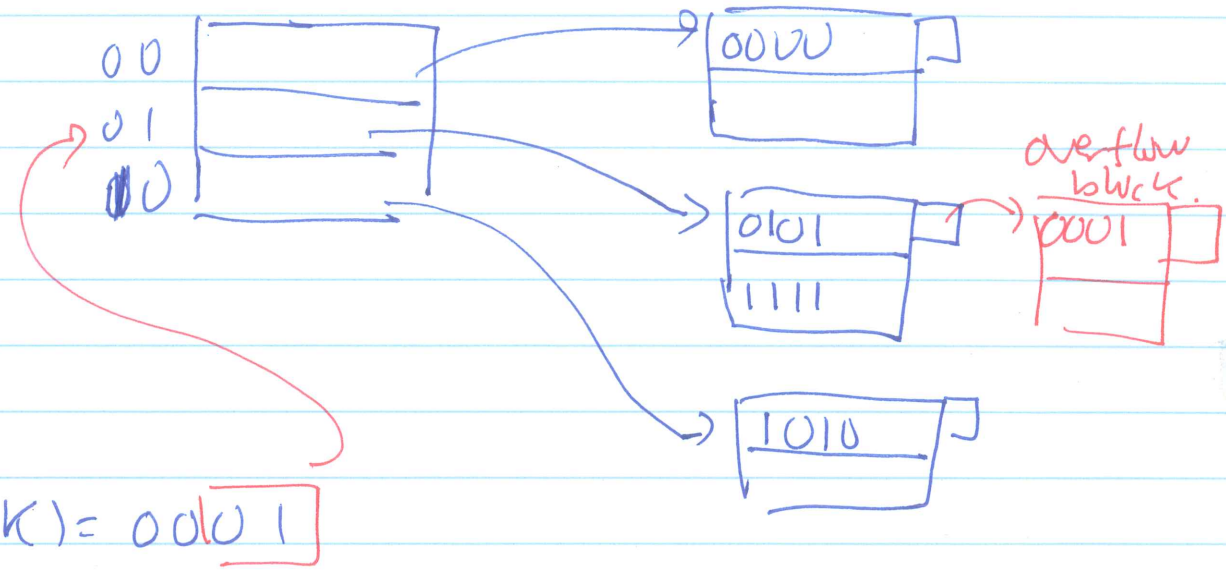


Note: 1111 is still ok.
Can be found, because:

11 is not a real bucket
→ use bucket 01 !!

Continued: Insert K, such that $h(K) = 0001$

Before insert: $i = 2$, $n = 3$, $r = 4$ $\quad \tau = 0.85$



$h(K) = 0001$

(Insertion: read things).

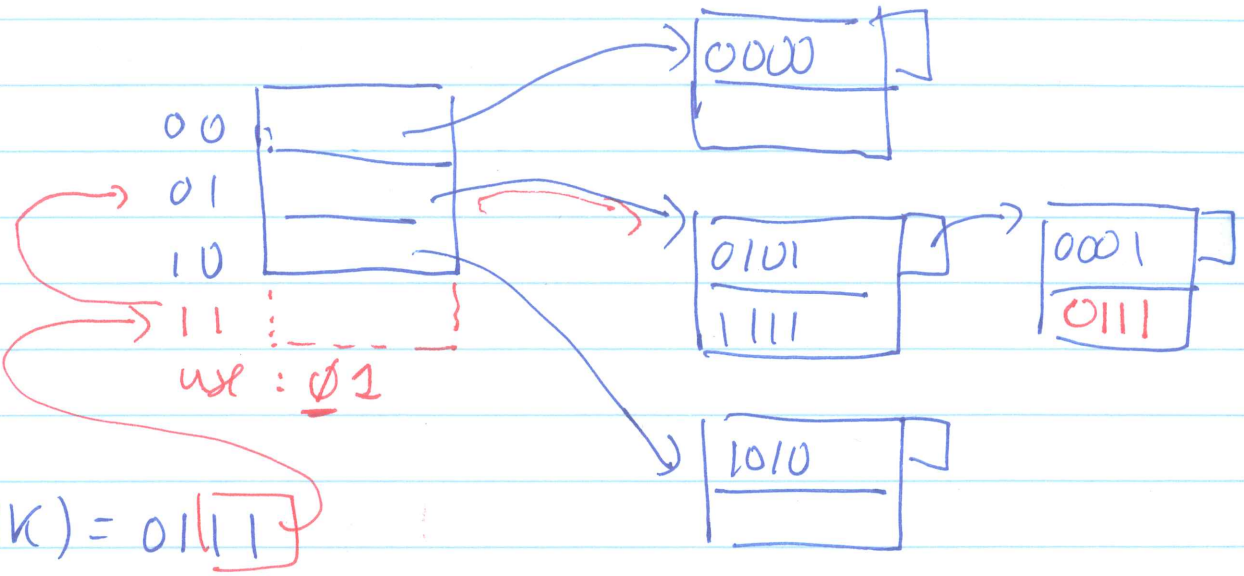
Check:

$$\frac{r}{n\gamma} = \frac{5}{3.2} = \frac{5}{6} = 0.83 \leq \tau.$$

No need to add another bucket!

Example continues: Insert K, with $h(K) = 0111$

Before Insertion: $i = 2$ $\quad n = 3$ $\quad r = 5$ $\qquad \tau = 0.55$



(Insertion: read things).

Check:

$$\frac{r}{n} = \frac{6}{3.2} = 1 \geq \tau$$

Add (bucket): 1 1

Split (bucket): 0 1

Result after splitting bucket 01:

$i = 2$ $\quad\quad$ $r = 0.85$

$n = 3$

$r = 6$