Hashing

Hash Tables

Hash function h: search key -> [0, ..., B-1].

Buckets are blocks, numbered [0, ..., B-1].

General idea: If a record with search key K exists, then it must be in bucket h(K).

- Cuts search down by a factor of B.
- One disk I/O if there is only one block per bucket.

Hash Table Operations

HashTable Lookup

- For record(s) with search key K, compute h(K); search that bucket.

HashTable Insertion

- Put in bucket h(K) if it fits; otherwise create an overflow block.
- Overflow block(s) are part of bucket.

HashTable Deletion

- Compute h(K); search bucket for record(s) with key K and delete entry

Example with 2 Records/Bucket

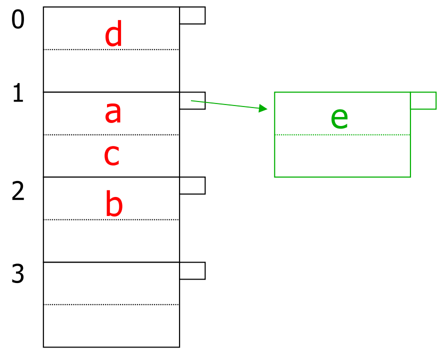
INSERT:

h(a) = 1

h(b) = 2

h(c) = 1

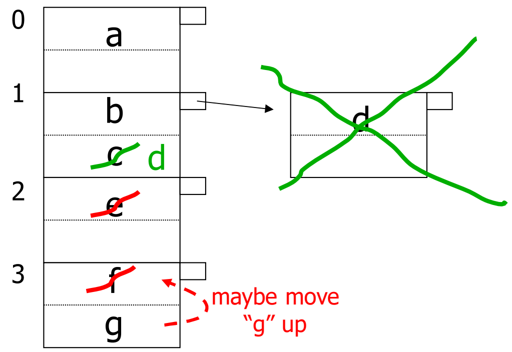
h(d) = 0



h(e) = 1

Example: Deletion

Delete:
e
f
c



How full should a Block be?

Try to keep space utilization

between 50% and 80%

Utilization = $\frac{\text{\# keys used}}{\text{total \# keys that fit}}$

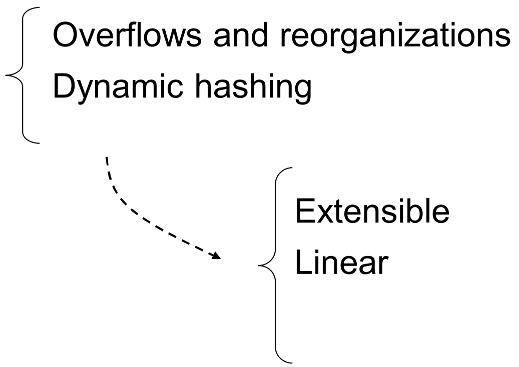
If < 50%, wasting space

If > 80%, overflows significant

depends on how good hash function is and on # keys/bucket

r/n

How do we cope with growth?



Overflows and reorganizations
Dynamic hashing

Extensible
Linear

Dynamic Hashing Framework

Hash function h produces a sequence of bits.

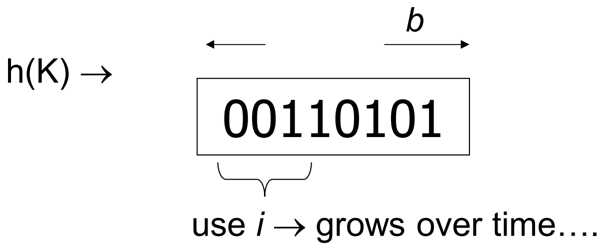
Only some of the bits are used at any time to determine placement of keys in buckets.

Extensible Hashing

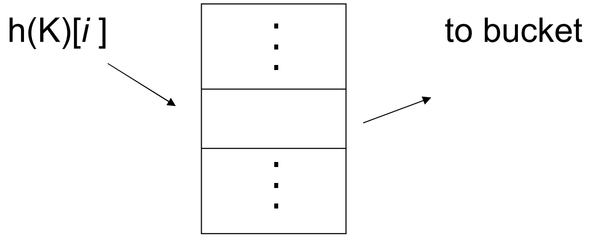
- Keep parameter i = number of bits from the beginning of h(K) determine the bucket.
- Bucket array now = pointers to blocks.
- A block can serve as several buckets.
- For each block, a parameter j <= i tells how many bits of h(K) determine membership in the block.
- I.e., a block represents 2^{i-j} buckets that share the first j bits of their number.

Extensible hashing: two ideas

(a) Use *i* of *b* bits output by hash function



(b) Use directory



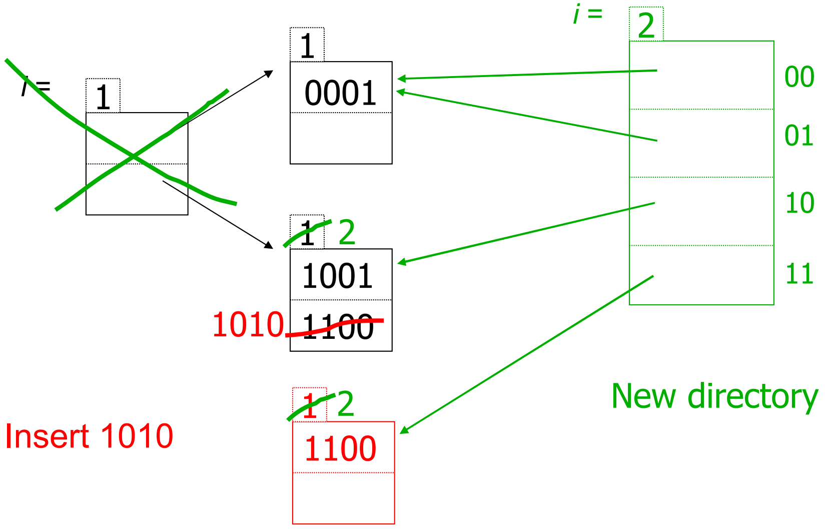
Extensible Hashtable Insert

If record with key K fits in the block pointed to by h(K), put it there.

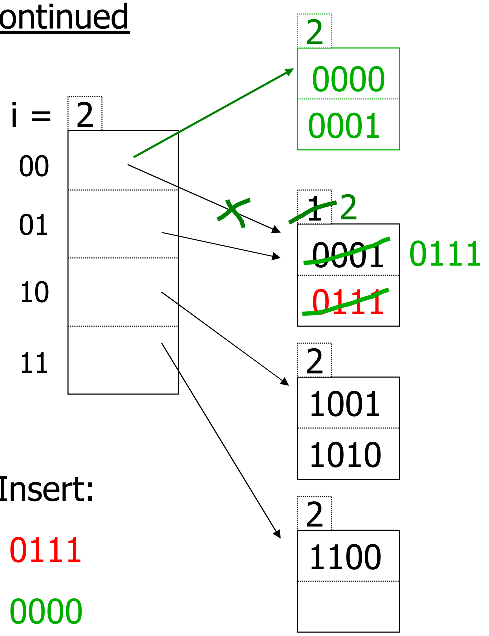
If not, let this block represent j bits.

- Case 1: j < i: Split block according to (j + 1)st bit; set j := j + 1.
- Case 2: j = i: Set i := i + 1; split bucket array; proceed as in (1).

Example: h(k) is 4 bits; 2 records/block



Example continued

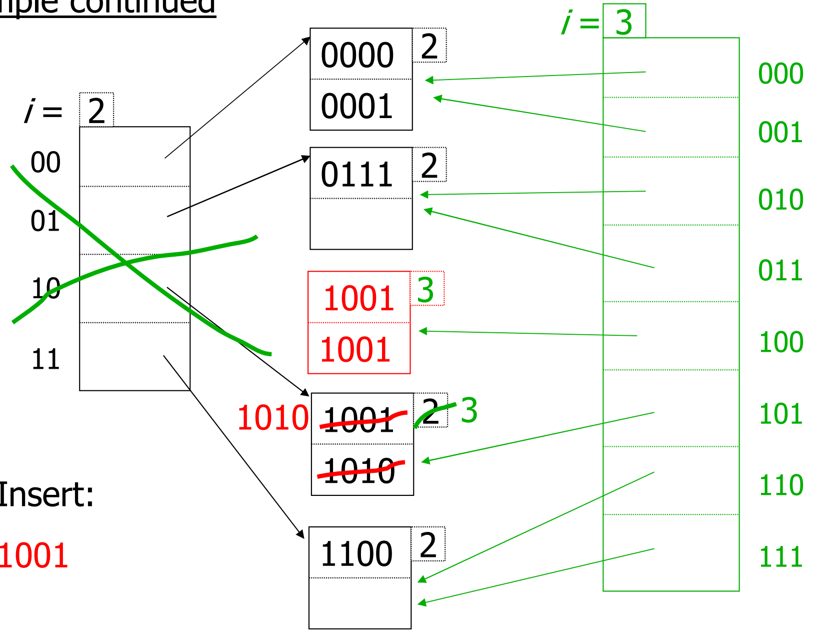


Insert:

0111

0000

Example continued



Insert:

1001

Summary Extensible Hashing

+ Can handle growing files
 - with less wasted space
 - with no full reorganizations

- Indirection
 (Not bad if directory in memory)

- Directory doubles in size

Linear Hashing

Use i bits from right (loworder) end of h(K).

Buckets numbered [0, ..., n-1], where $2^{i-1} < n <= 2^i$.

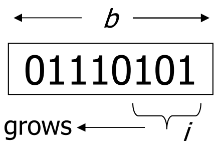
Let the last i bits of h(K) be m = $(a_1a_2...a_i)$.

- 1. If m < n, then record belongs in bucket m.
- 2. If $n <= m < 2^i$, then record belongs in bucket $m - 2^{i-1}$.

Difference to Extensible Hashing

Two ideas:

(a) Use *i* low order bits of hash



(b) File grows linearly

Utilization Limit: A threshold is set for the utilization of the hash table.
Adding a Bucket (n := n + 1): When an insertion causes the utilization to exceed the limit, a new bucket is added to the hash table. This increases the total number of buckets (n).

Linear HashTable Insert

Utilization is defined as the total number of hash entries, divided by the number of possible entries in the primary blocks of hash buckets (i.e., not counting the capacity of overflow blocks!)

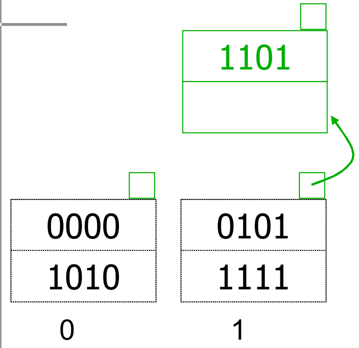
Pick an upper limit on utilization for the hash-table

If an insertion exceeds this utilization limit, add a bucket, i.e., set n := n + 1.

- If new n is $2^i + 1$, set i := i + 1. No change in bucket numbers needed --- just imagine a leading 0.
- Need to split bucket n - 2^{i-1} because there is now a bucket numbered (old) n.

Note, that the bucket that is split need not be one with a large number of records!

Example *b*=4 bits, *i* =1, 2 records/block



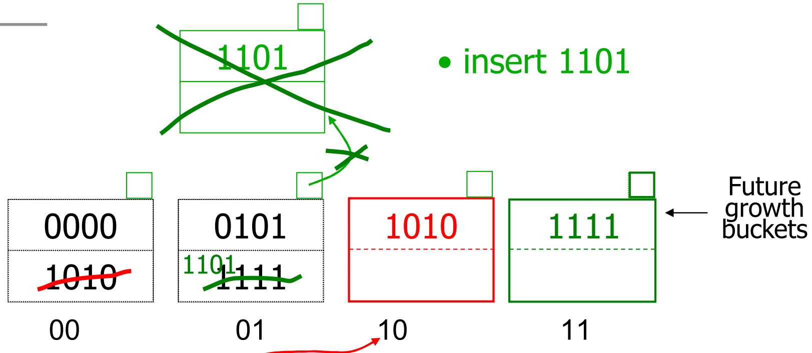
- insert 1101
- can have overflow chains!

n-1 = 1 (max used block)

Rule

If h(k)[*i*] ≤ *n-1*, then
look at bucket h(k)[i]
else, look at bucket h(k)[*i*] - 2^{i-1}

Example *b*=4 bits, *i* =2, 2 records/block

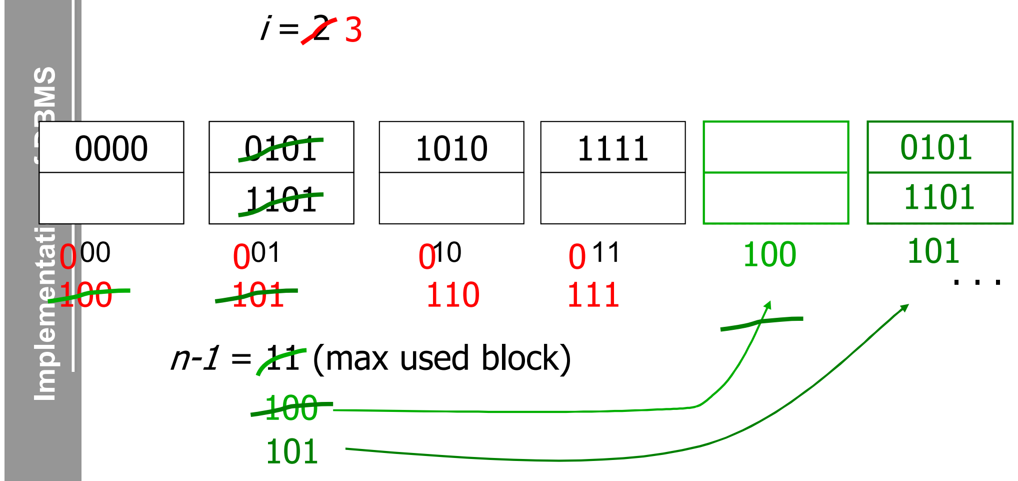


- insert 1101

n-1 = ~~01~~ (max used block)
~~10~~
11

Growth:
i has increased to 2, indicating that the hash table has grown.
New blocks (1010 and 1111) have been added.
The maximum used block is now n-1 = 01 (in binary).

Example Continued: How to grow beyond this?



Summary Linear Hashing

+ Can handle growing files

- with less wasted space
- with no full reorganizations

+ No indirection like extensible hashing

- Can still have overflow chains
split might not improve situation for full buckets

Indexing vs Hashing

Hashing good for queries looking for a particular search key

```
e.g.,   SELECT …
              FROM R
              WHERE R.A = 5
```

Classical indexing and B-Trees good for

Range Searches:

```
e.g.,   SELECT
              FROM R
              WHERE R.A > 5
```