CS411
Database Systems

07: Indexing

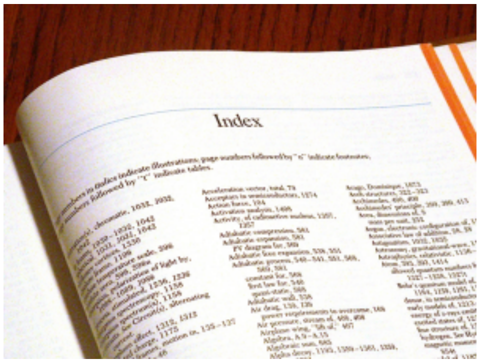
14.1-14.3

Why Do We Learn This?

Q: What is “indexing”?

- What is an index?
- To build an index.
- To maintain an index

What is an index?



Indexes in databases

- An *index* on a file speeds up selections on the *search key field(s)*
- Search key = any subset of the fields of a relation
  - *Search key* is not the same as *key* (minimal set of fields that uniquely identify a record in a relation).
- Entries in an index: (k, r), where:
  - k = the key
  - r = the record OR record id OR record ids

Some terminology

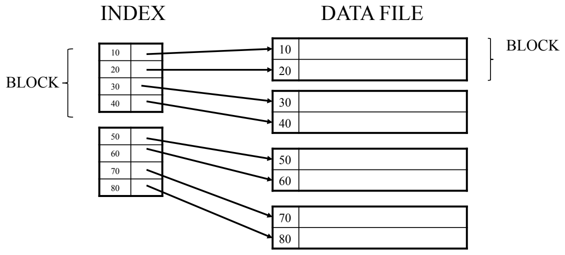
- *Data file*: has the data corresponding to a relation
- *Index file*: has the index
- File consists of smaller units called “blocks” (e.g. of size 4 KB or 8 KB)

Types of Indexes

- Clustered/unclustered
  - Clustered = records sorted in the key order
  - Unclustered = no
- Dense/sparse
  - Dense = each record has an entry in the index
  - Sparse = only some records have
- Primary/secondary
  - Primary = on the primary key
  - Secondary = on any key
  - Some textbooks interpret these differently
- B+ tree / Hash table / …

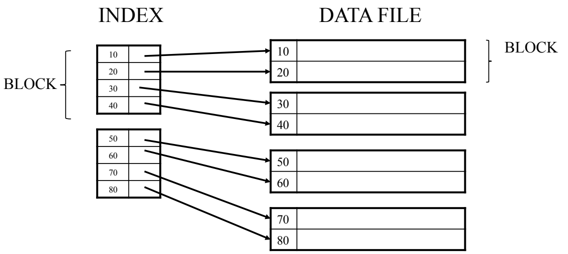
Ex: Clustered, Dense Index

- Clustered: File is sorted on the index attribute
- *Dense*: sequence of (key,pointer) pairs



Ex: Clustered, Dense Index

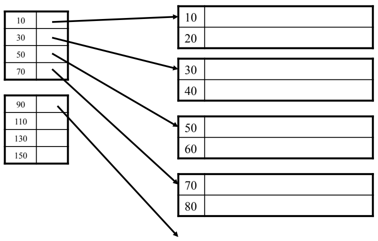
- Clustered: File is sorted on the index attribute
- *Dense*: sequence of (key,pointer) pairs



• Index blocks are much smaller than data blocks. Index may even fit into main memory

Clustered, Sparse Index

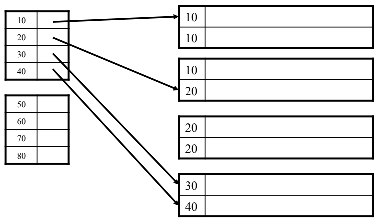
- *Sparse* index: one key per data block, corresponding to the lowest search key in that block

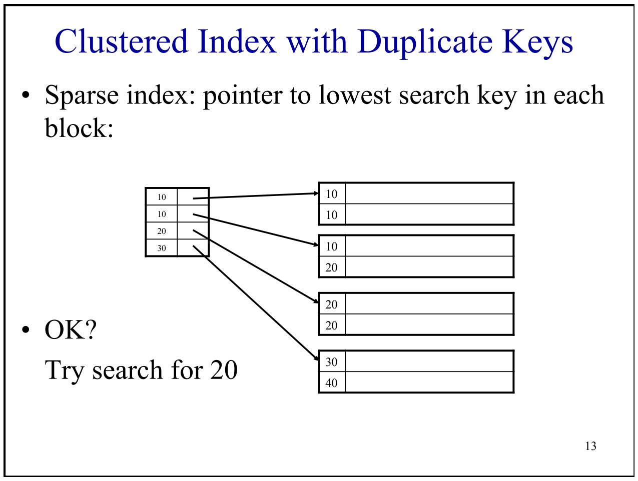


What if there are duplicate keys?

Clustered Index with Duplicate Keys

- Dense index: point to the first record with that key



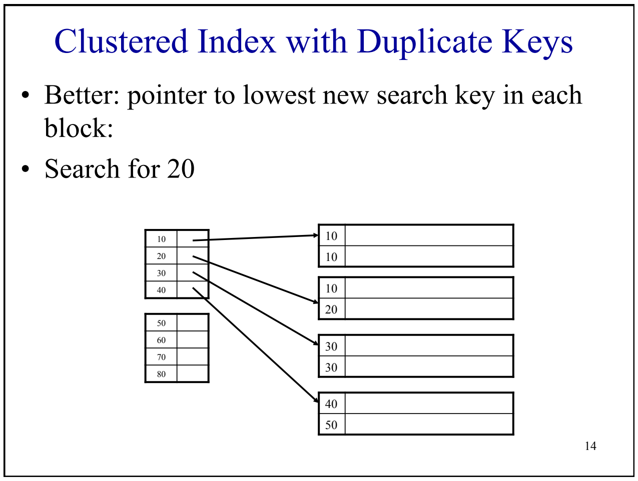


Clustered Index with Duplicate Keys

- Sparse index: pointer to lowest search key in each block:
- OK?

  Try search for 20

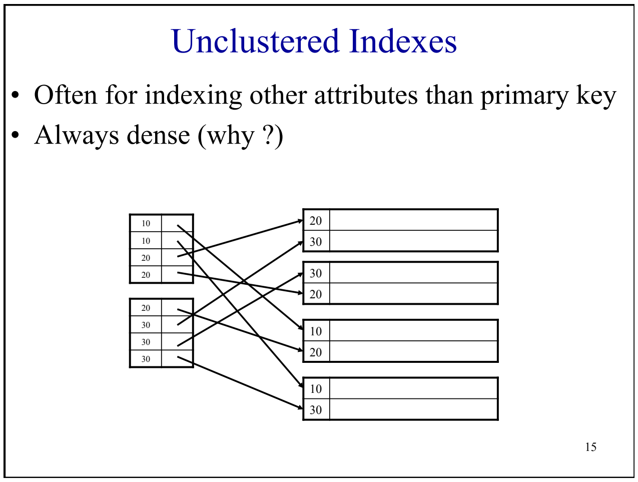
13



Clustered Index with Duplicate Keys

- Better: pointer to lowest new search key in each block:
- Search for 20

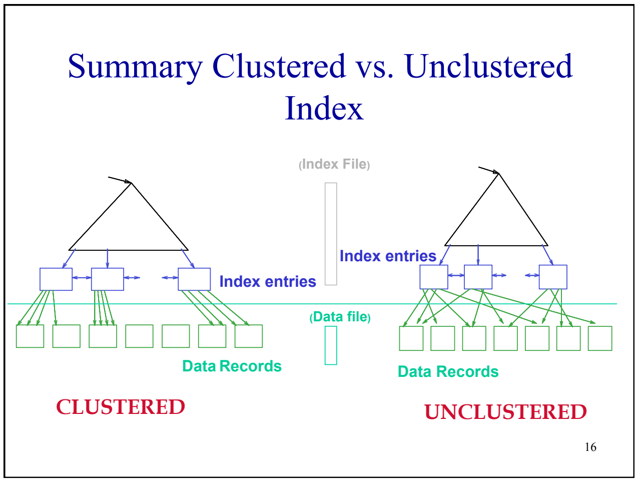
14



Unclustered Indexes

- Often for indexing other attributes than primary key
- Always dense (why ?)

15



Summary Clustered vs. Unclustered Index

16

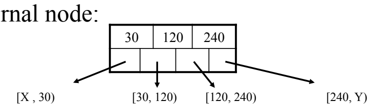
B+ Trees

B-Trees/B+Trees: B__?__?___ Trees

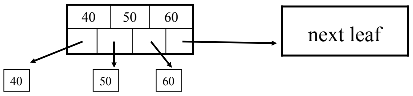
- Intuition:
  - The index can be very large.
  - Index of index?
  - Index of index of index?
  - How best to create such a multi-level index?
- B+trees:
  - Textbook refers to B+trees (a popular variant) as B-trees (as most people do)
  - Distinction will be clear later

B+ Trees Basics

- Parameter d = the *degree*
- Each node has [d, 2d] keys (except root)
  - Internal node:



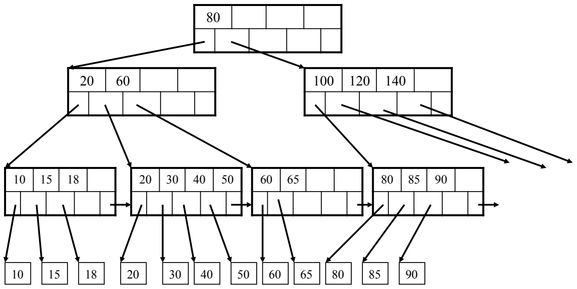
  - Leaf:



Data records:

B+ Tree Example

d = 2



B+ Tree Design

- How large is d ?
- Example:
  - – Key size = 4 bytes
  - – Pointer size = 8 bytes
  - – Block size = 4096 byes
- 2d x 4 + (2d+1) x 8 <= 4096
- d = 170

Searching a B+ Tree

- Exact key values:
  - – Start at the root
  - – Proceed down, to the leaf
- Range queries:
  - – As above
  - – Then sequential traversal
  - – This is where the “next leaf” pointer is useful

```
Select name
From people
Where age = 25
```

```
Select name
From people
Where 20 <= age
  and  age <= 30
```

Some applications of B+ trees

1. Search key is primary key; index is dense. Data file may or may not be sorted by key.

2. Data file is sorted by primary key; index is sparse.

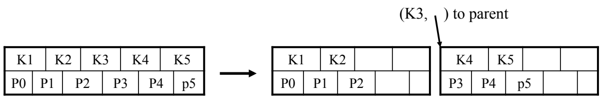
B+ Trees in Practice

- Typical d: 100. Typical fill-factor: 67%.
  - – average “fanout” = 133
- Typical capacities:
  - – Height 4: 133^4 = 312,900,700 records
  - – Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool:
  - – Level 1 = 1 page = 8 Kbytes
  - – Level 2 = 133 pages = 1 Mbyte
  - – Level 3 = 17,689 pages = 133 MBytes

Insertion in a B+ Tree

Assume dense index.

Insert (K, P)
- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:

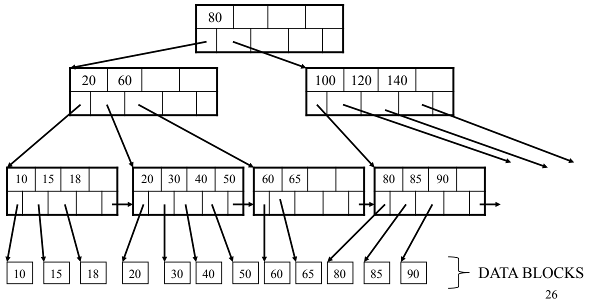


- If leaf, keep K3 too in right node
- When root splits, new root has 1 key only
  - that's why root is special for degree satisfaction

Insertion in a B+ Tree

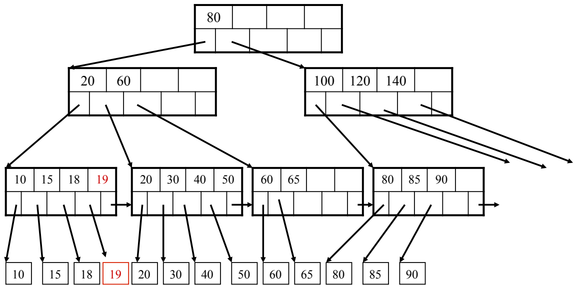
Assume d=2.

Insert K=19



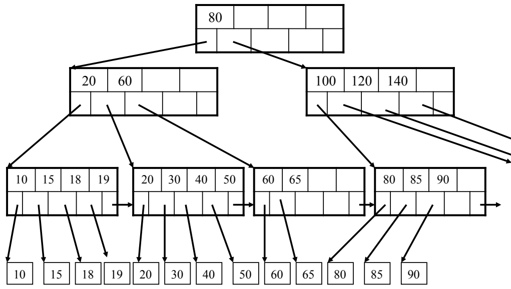
Insertion in a B+ Tree

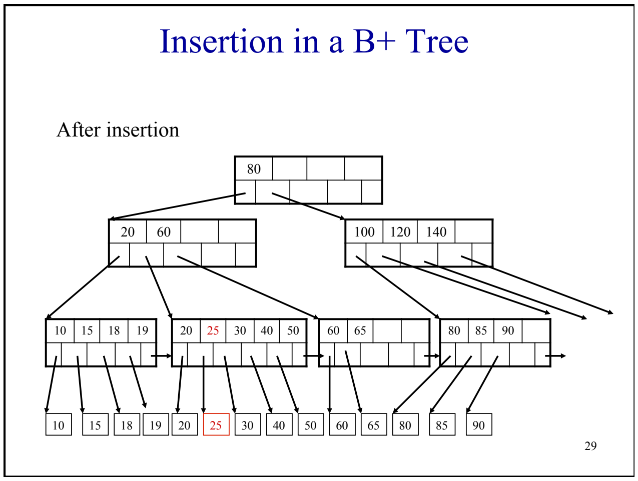
After insertion



Insertion in a B+ Tree

Now insert 25

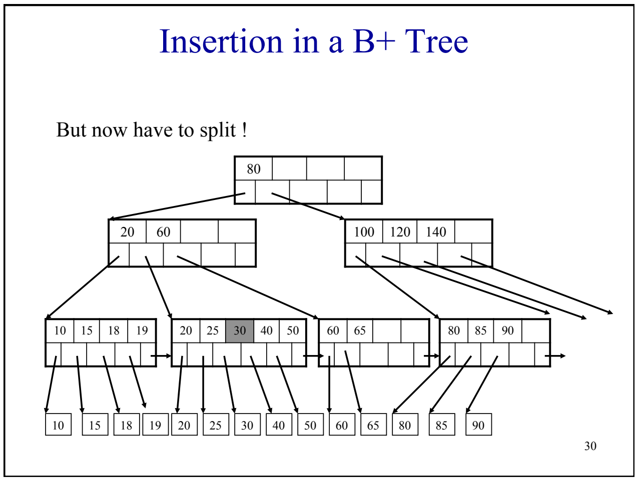




Insertion in a B+ Tree

After insertion

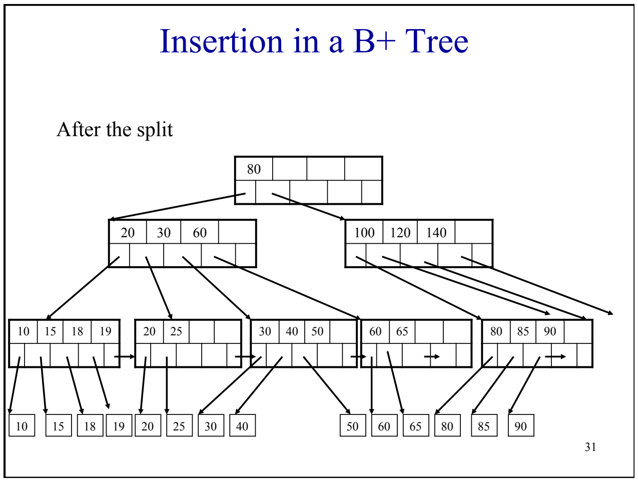
29



Insertion in a B+ Tree

But now have to split !

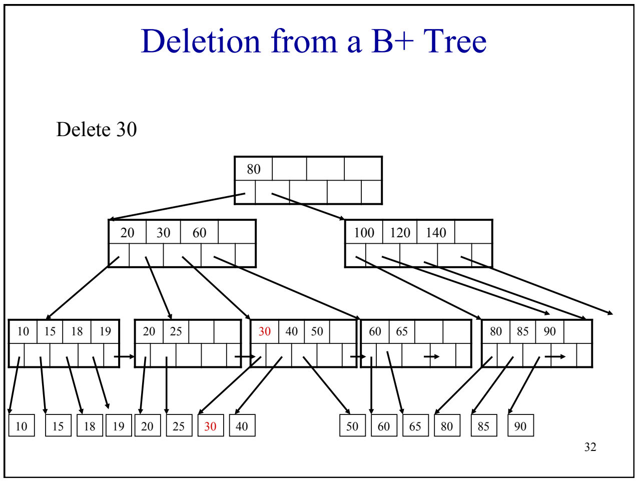
30



Insertion in a B+ Tree

After the split

31



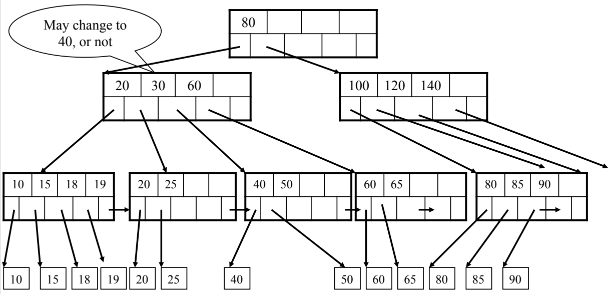
Deletion from a B+ Tree

Delete 30

32

Deletion from a B+ Tree

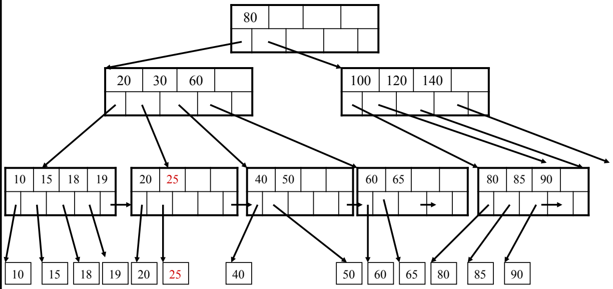
After deleting 30



May change to 40, or not

Deletion from a B+ Tree

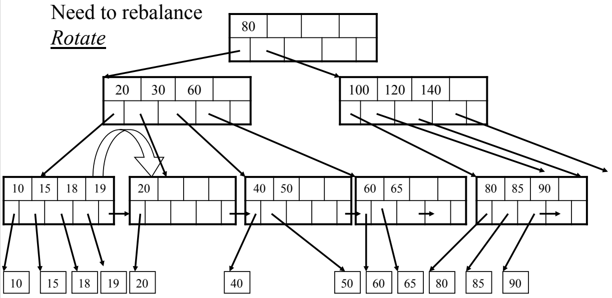
Now delete 25



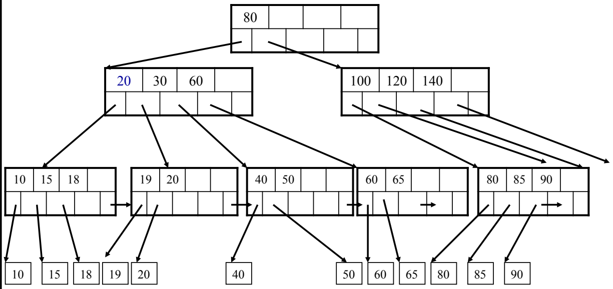
Deletion from a B+ Tree

Rotation in general can involve either sibling, but here only the left sibling can help

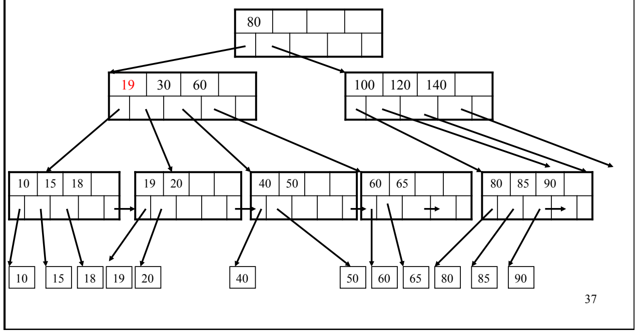
After deleting 25
Need to rebalance
Rotate



Deletion from a B+ Tree

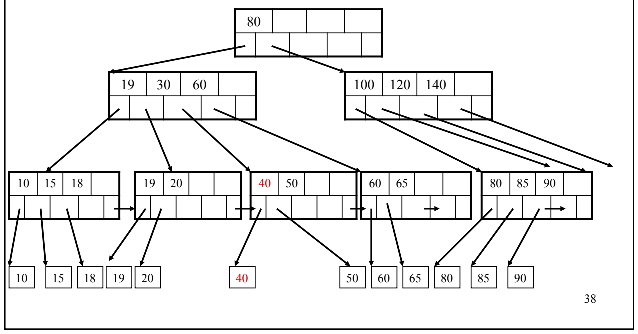


Deletion from a B+ Tree



Deletion from a B+ Tree

Now delete 40

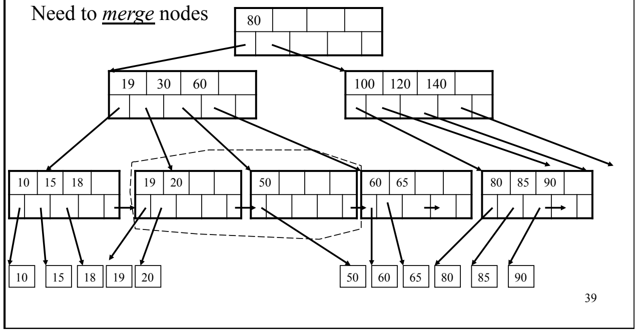


Deletion from a B+ Tree

After deleting 40

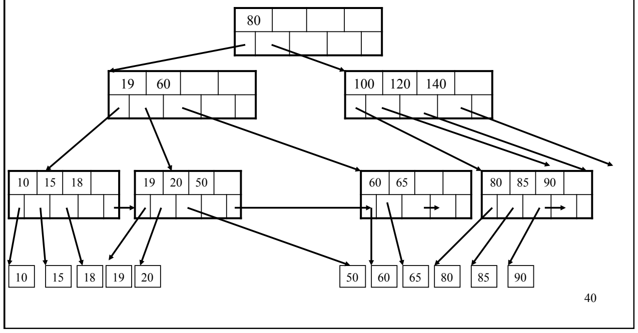
Rotation not possible

Need to *merge* nodes



Deletion from a B+ Tree

Final tree



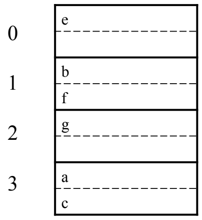
Hash Tables

Hash Tables

- Secondary storage hash tables are much like main memory ones
- Recall basics:
  - There are n ***buckets***
  - A hash function f(k) maps a key k to {0, 1, …, n-1}
  - Store in bucket f(k) a pointer to record with key k
- Secondary storage: bucket = block
  - Store in bucket f(k) any record with key k
  - use overflow blocks when needed

Hash Table Example

- Assume 1 bucket (block) stores 2 records
- h(e)=0
- h(b)=h(f)=1
- h(g)=2
- h(a)=h(c)=3

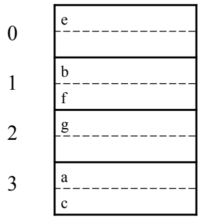


Searching in a Hash Table

- Search for a:
- Compute h(a)=3
- Read bucket 3
- 1 disk access

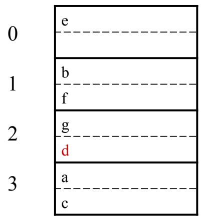
HOW ?

Main memory may have an array of pointers (to buckets) accessible by bucket number.



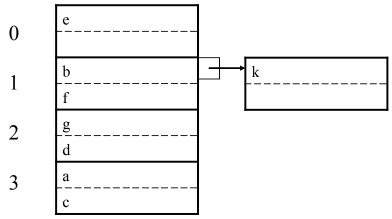
Insertion in Hash Table

- Place in right bucket, if space
- E.g. h(d)=2



Insertion in Hash Table

- Create overflow block, if no space
- E.g. h(k)=1



- More overflow blocks may be needed

Hash Table Performance

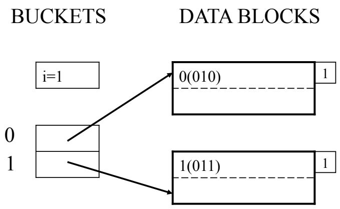
- Excellent, if no overflow blocks
- Degrades considerably when many overflow blocks.

Extensible Hash Table

- Allows hash table to grow, to avoid performance degradation
- Assume a hash function h that returns numbers in $\{0, \ldots, 2^k - 1\}$
- Start with $n = 2^i << 2^k$, only look at first i most significant bits

Extensible Hash Table

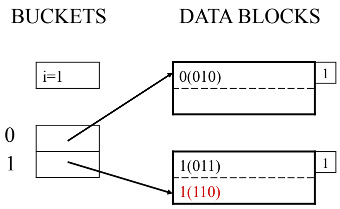
- E.g. i=1, n=2, k=4



- Note: we only look at the first bit (0 or 1)

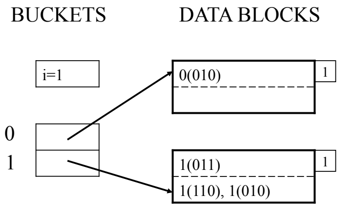
Insertion in Extensible Hash Table

- Insert 1110



Insertion in Extensible Hash Table

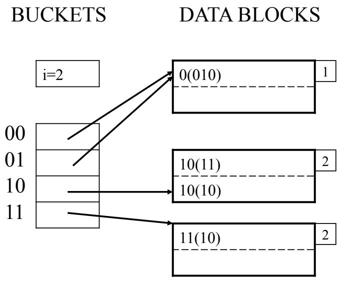
- Now insert 1010



- Need to split block and extend bucket array
- i becomes 2

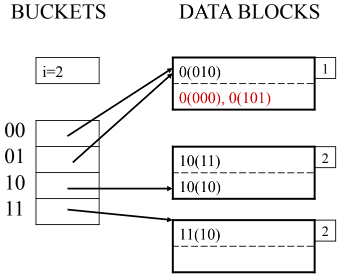
Insertion in Extensible Hash Table

- Now insert 1010 (cont.)



Insertion in Extensible Hash Table

- Now insert 0000, then 0101



BUCKETS

DATA BLOCKS

i=2

00

01

10

11

0(010)

0(000), 0(101)

1

10(11)

10(10)

2

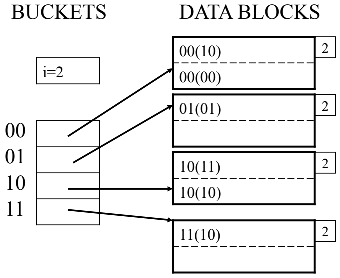
11(10)

2

- Need to split block, but not bucket array

Insertion in Extensible Hash Table

- After splitting the block



BUCKETS

DATA BLOCKS

i=2

00

01

10

11

00(10)

00(00)

2

01(01)

2

10(11)

10(10)

2

11(10)

2

Performance Extensible Hash Table

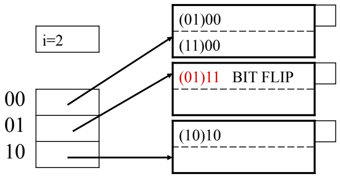
- No overflow blocks: access always one read
- BUT:
  - Extensions can be costly and disruptive
  - After an extension table may no longer fit in memory
  - Imagine three records whose keys share the first 20 bits. These three records cannot be in same block (assume two records per block). But a block split would require setting i = 20, i.e., accommodating for 2^20 = 1 million buckets, even though there may be only a few hundred records.

Linear Hash Table

- Idea: add only one bucket at a time
- Problem: n = no longer a power of 2
- Let i be #bits necessary to address n buckets.
  - $2^{i-1} < n <= 2^{i}$
- After computing h(k), use last i bits:
  - If last i bits represent a number >= n, change msb from 1 to 0 (get a number < n)
- Also, allow overflow blocks

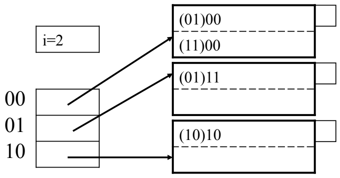
Linear Hash Table Example

- N=3



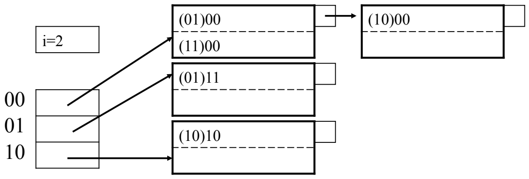
Linear Hash Table Example

- Insert 1000:



Linear Hash Table Example

- Insert 1000: overflow blocks…

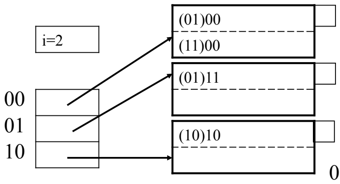


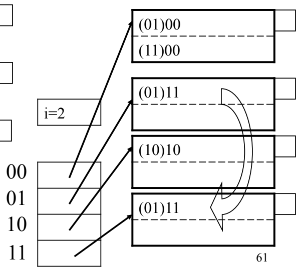
Linear Hash Tables

- Extend n:=n+1 when average number of records per block exceeds (say) 80%

Linear Hash Table Extension

- From n=3 to n=4



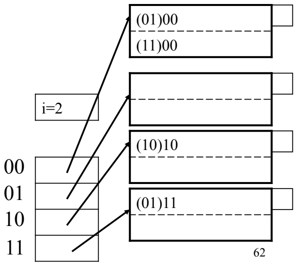


- Only need to touch one block (which one ?)

Linear Hash Table Extension

- From n=3 to n=4 finished

- Extension from n=4 to n=5 (new bit)
- Need to touch every single block (why ?)



- See examples in text.