Today:

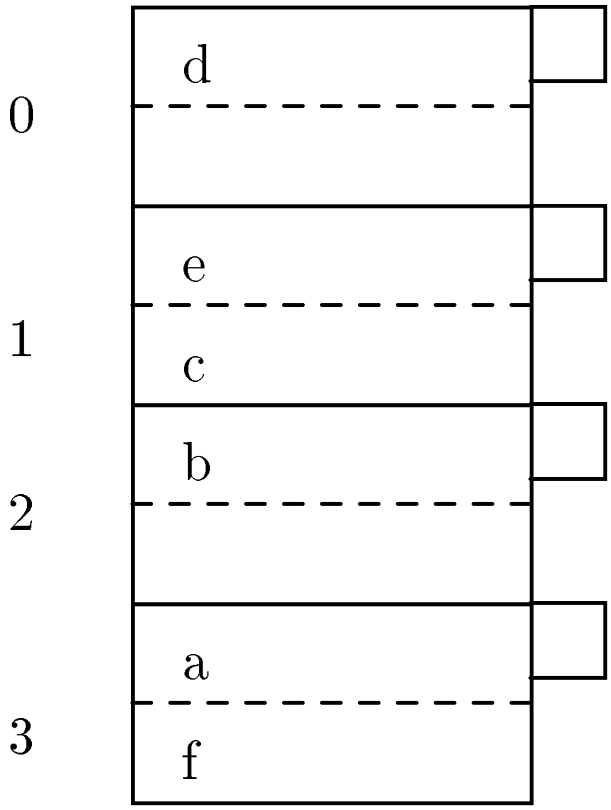
- Hashing; static and dynamic hash tables.
- Begin study of multidimensional data.
  - Difficult queries; point and region data.
  - Multilevel indexing, kd-trees.

Soon:

- Finish multidimensional data structures.
  - Grid-file, partitioned hashing, R-trees.
- Then — on to query optimization.

Hash Tables

- *Hash function* h: search key $\rightarrow$ $[0..B-1]$.
- *Buckets* are blocks, numbered $[0..B-1]$.
- Big 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 Lookup

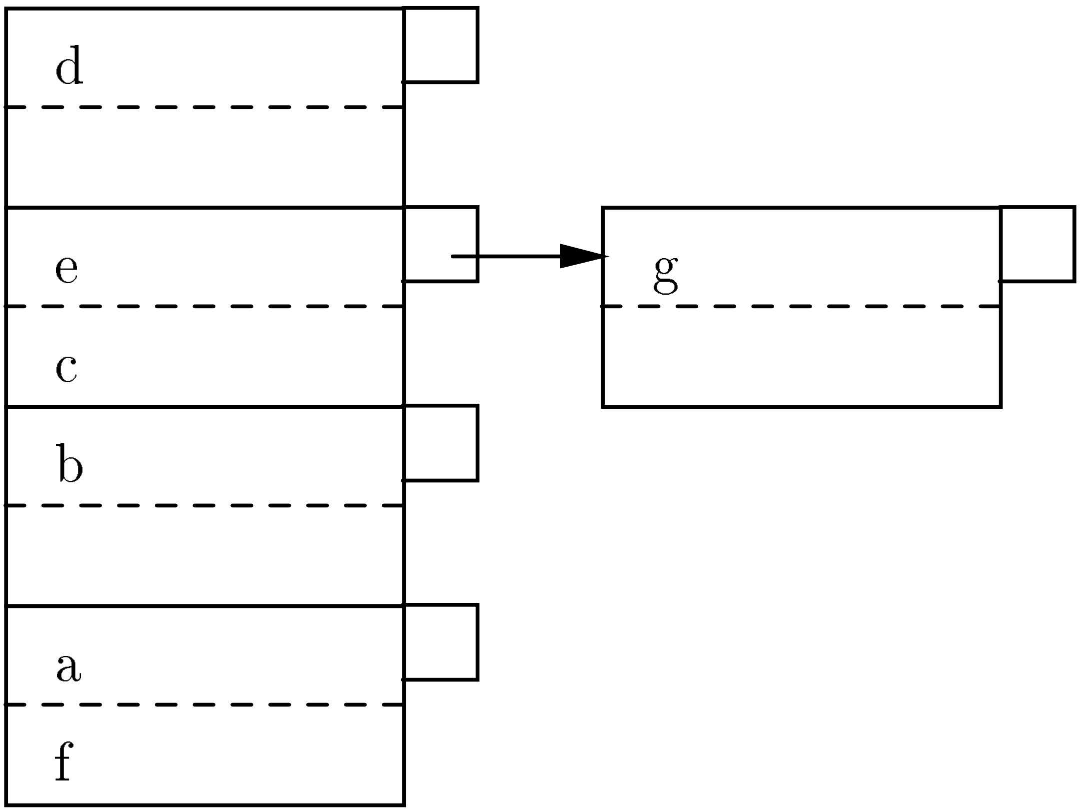
- For record(s) with search key K, compute $h(K)$; search that bucket.

Hash-Table Insertion

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

Example

Insert record with search key g.

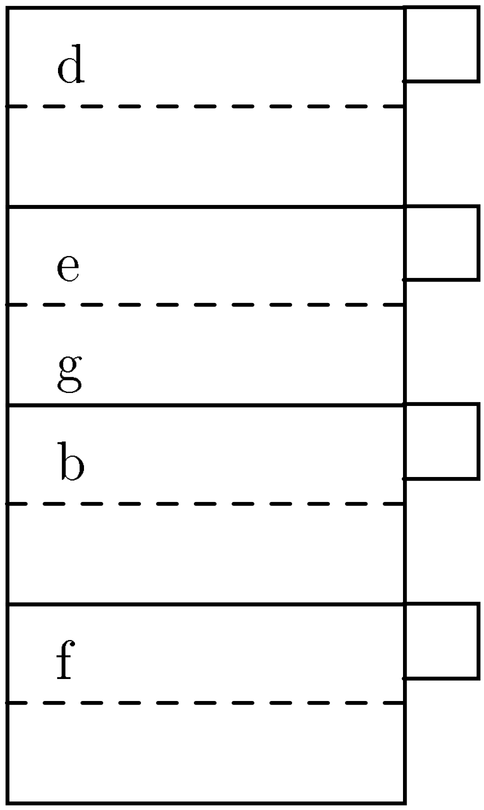


Hash-Table Deletion

- Compute $h(K)$; search bucket for record(s) with key K.

Example

Delete record with key c.



What if the File Grows too Large?

- Efficiency is highest if

  $$\#\text{records} < \#\text{buckets} \times \#(\text{records/block})$$

- If file grows, need *dynamic hashing* method to maintain above relationship.
  - *Extensible Hashing*: double the number of buckets when needed.
  - *Linear hashing*: add one more bucket as appropriate.

Problem

How does hash-table access compare with B-tree access?

- (a) Records in buckets? (b) Buckets point to actual records?

Dynamic Hashing Framework

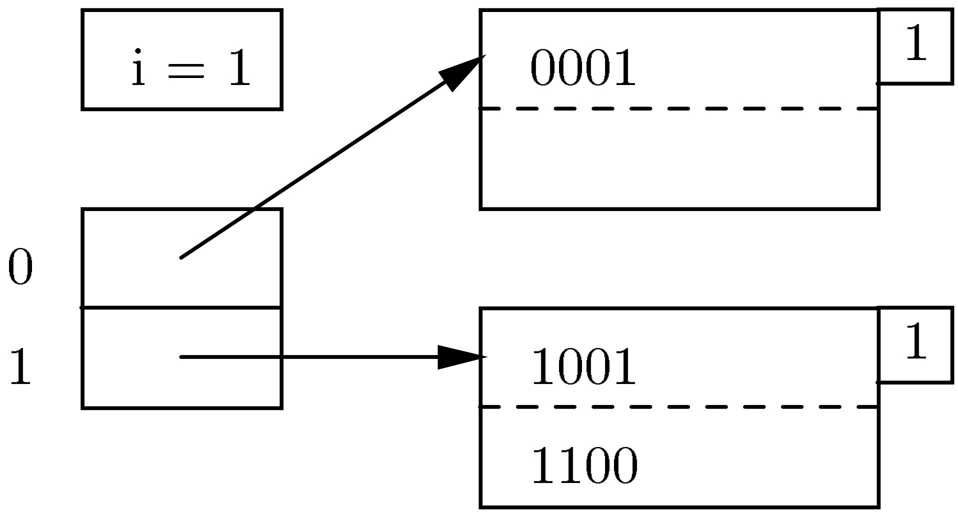
- Hash function h produces a sequence of k bits.
- Only some of the bits are used at any time to determine placement of keys in buckets.

Extensible Hashing

- Keep parameter i = number of bits from the beginning of $h(K)$ determine the bucket.
- Bucket array now = pointers to buckets.
  - A block can serve as several buckets.
  - For each block, a parameter $j \leq i$ 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.

Example

An extensible hash table when $i = 1$:

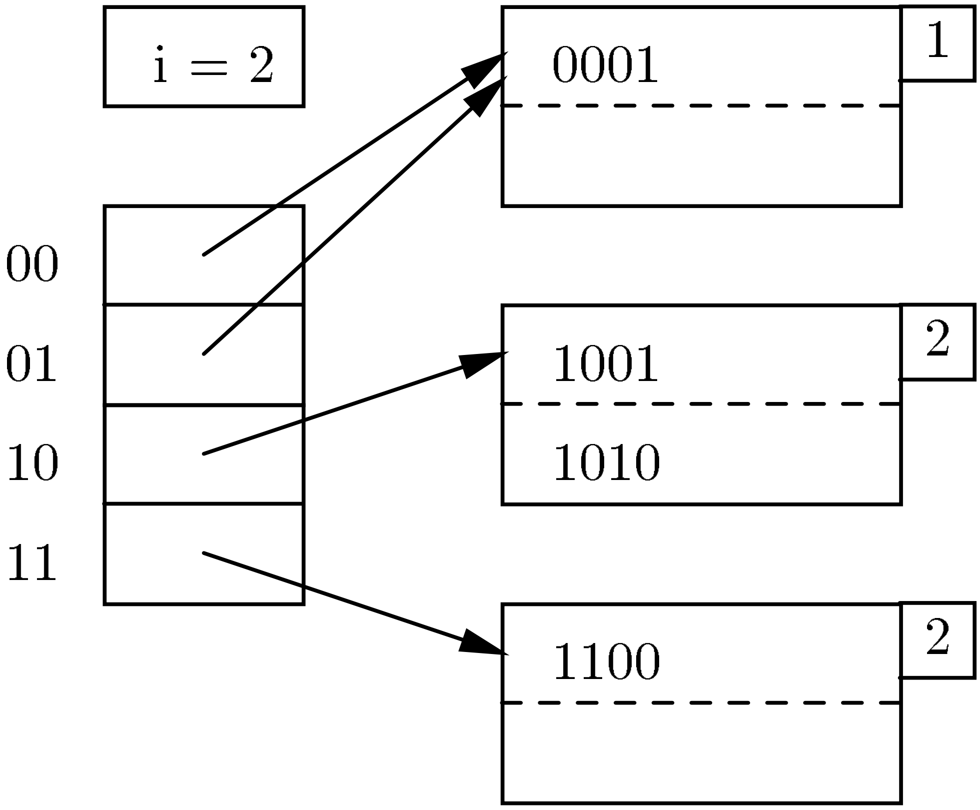


Extensible Hash-table Insert

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

Example

Insert record with $h(K) = 1010$.

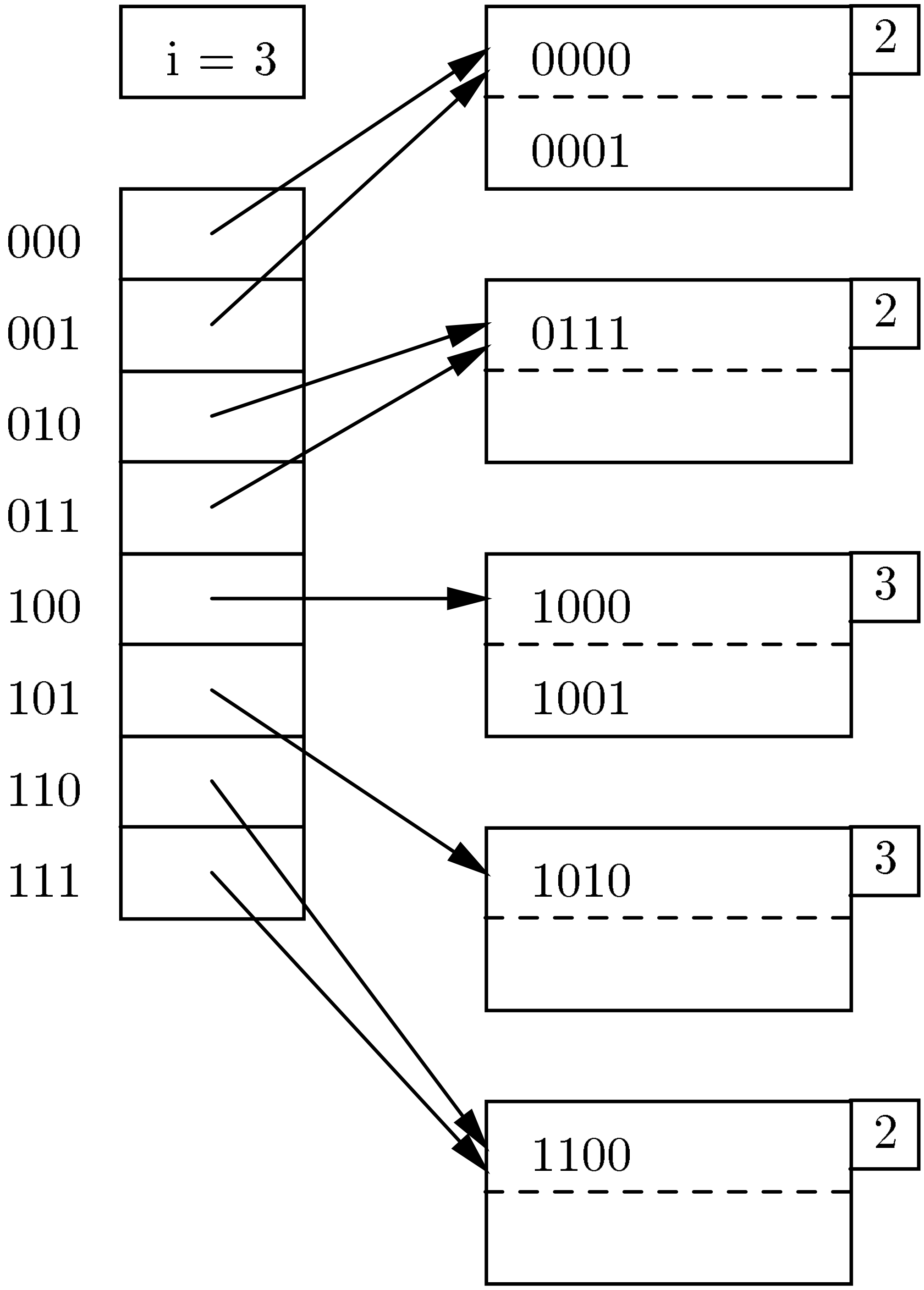


Next: records with $h(K) = 0000$; $h(K) = 0111$.

- Bucket for 0... gets split, but i stays at 2.

Then: record with $h(K) = 1000$.

- Overflows bucket for 10...
- Raise i to 3.

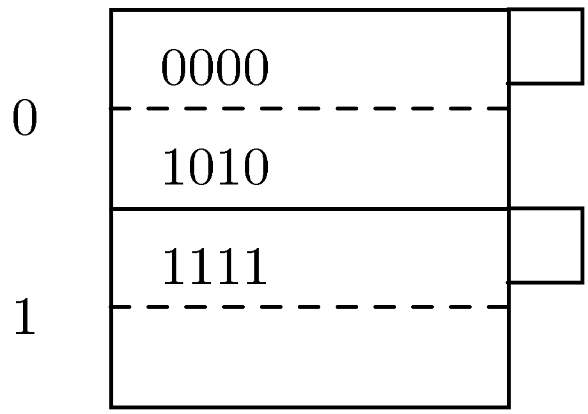

i = 3
000
001
010
011
100
101
110
111
0000
0001
2
0111
2
1000
1001
3
1010
3
1100
2

Linear Hashing

- Use i bits from right (low-order) end of $h(K)$.
- Buckets numbered $[0..n-1]$, where $2^{i-1} < n \leq 2^i$.
- Let last i bits of $h(K)$ be $m = (a_1 a_2 \cdots a_i)$.

    1. If $m < n$, then record belongs in bucket m.

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



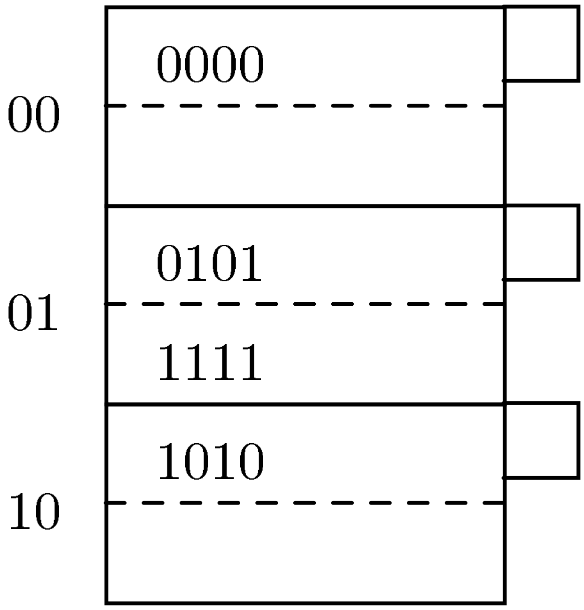
Linear Hash-Table Insert

- Pick an upper limit on capacity, e.g., 85% (1.7 records/bucket in our example).
- If an insertion exceeds capacity limit, $n := n + 1$.
  - ❖ 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.

Example

Insert record with $h(K) = 0101$.

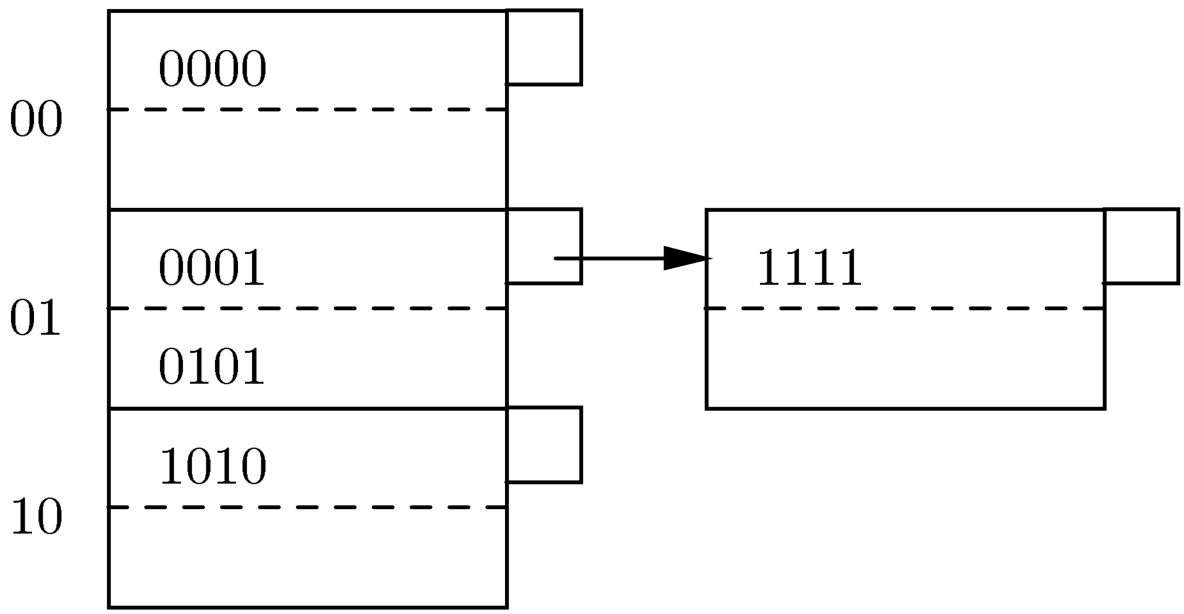
- Capacity limit exceeded; increment n.



Example

Insert record with $h(K) = 0001$.

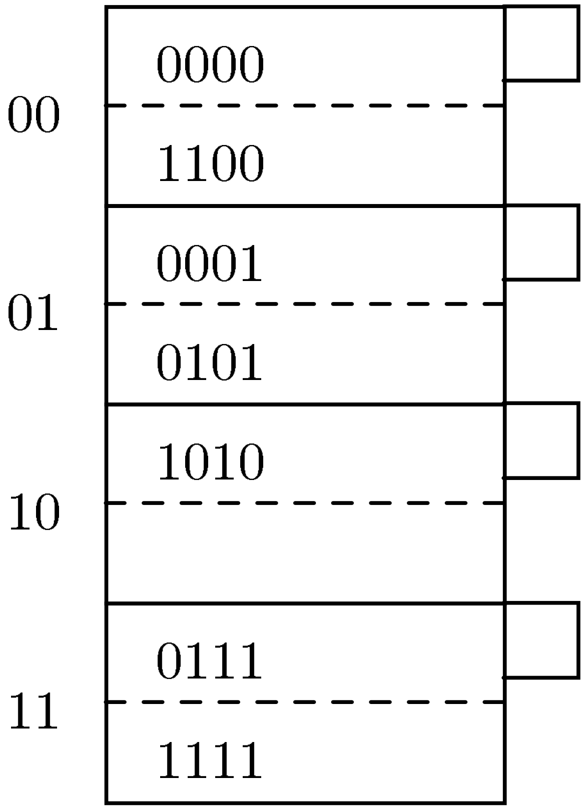
- Capacity limit not exceeded.
- But bucket is full; add overflow bucket.



Example

Insert record with $h(K) = 1100$.

- Capacity exceeded; set $n = 4$, add bucket 11.
- Split bucket 01.



Multidimensional Data

Many applications of databases are "geographic" = 2-dimensional data. Others involve large numbers of dimensions.

- Example: data mining information about sales.
  - ✦ A sale is described by (store, day, item, color, size, etc.). Sale = point in 5-dim space.
  - ✦ A customer is described by (age, salary, zipcode, marital-status, etc.).

Typical Queries

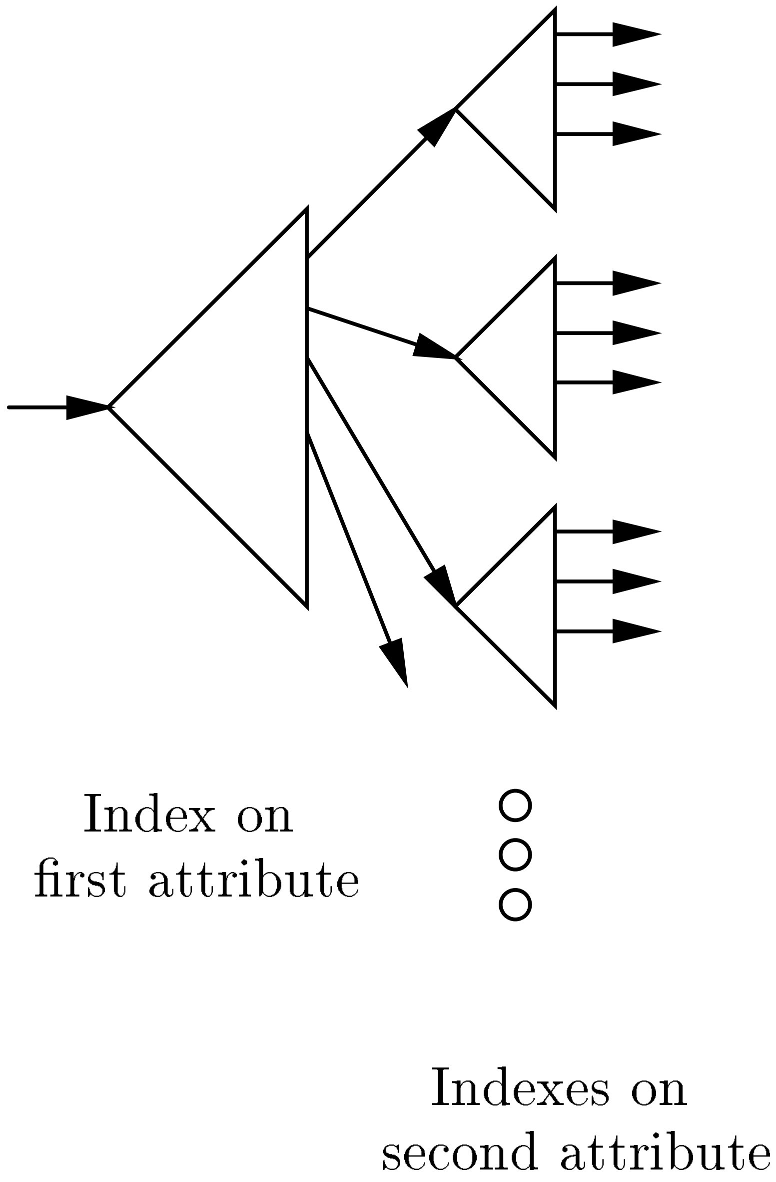
- *Range queries*: "How many customers for gold jewelry have age between 45 and 55, and salary less than 100K?"
- *Nearest neighbor*: "If I am at coordinates (x, y), what is the nearest McDonalds."
  - ✦ Expressible in SQL. Do you see how?

Big Impediment

For these types of queries, there is no clean way to eliminate lots of records that don't meet the condition of the `WHERE`-clause.

Approaches

1. Index on attributes independently.
   - Intersect pointers in main memory to save disk I/O.
   - **Problem:** Does this structure help with nearest-neighbor?
2. *Multiple key index*: Index on one attribute provides pointer to an index on the other.


Index on
first attribute
Indexes on
second attribute

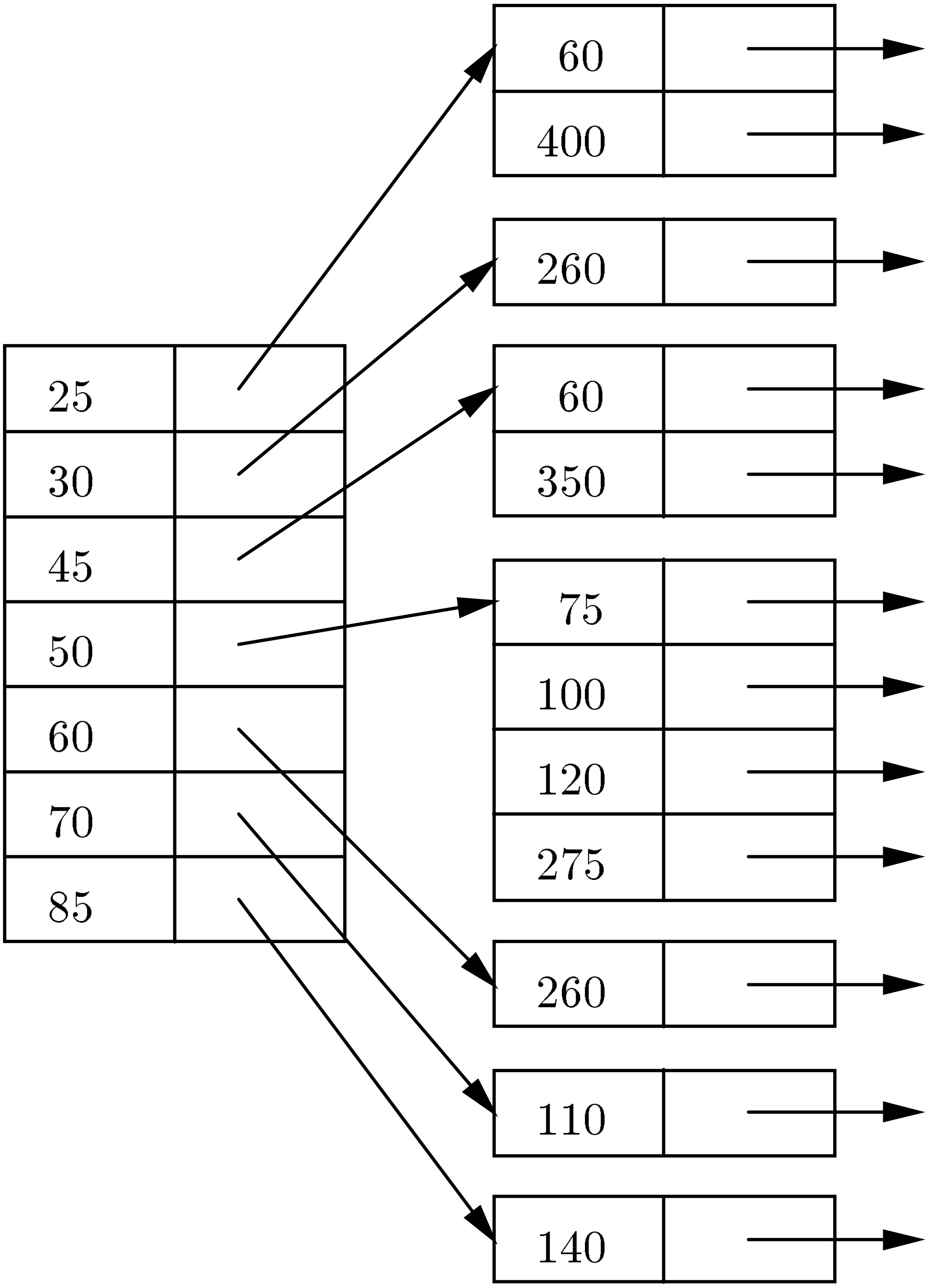
Example

"Who buys gold jewelry" (age and salary only). Raw data in age-salary pairs:

$$\begin{array}{llll} (25,60) & (45,60) & (50,75) & (50,100) \\ (50,120) & (70,110) & (85,140) & (30,260) \\ (25,400) & (45,350) & (50,275) & (60,260) \end{array}$$

Question

For what kinds of conditions about age and salary will a multiple-key index (age first) significantly reduce the number of disk I/O's?



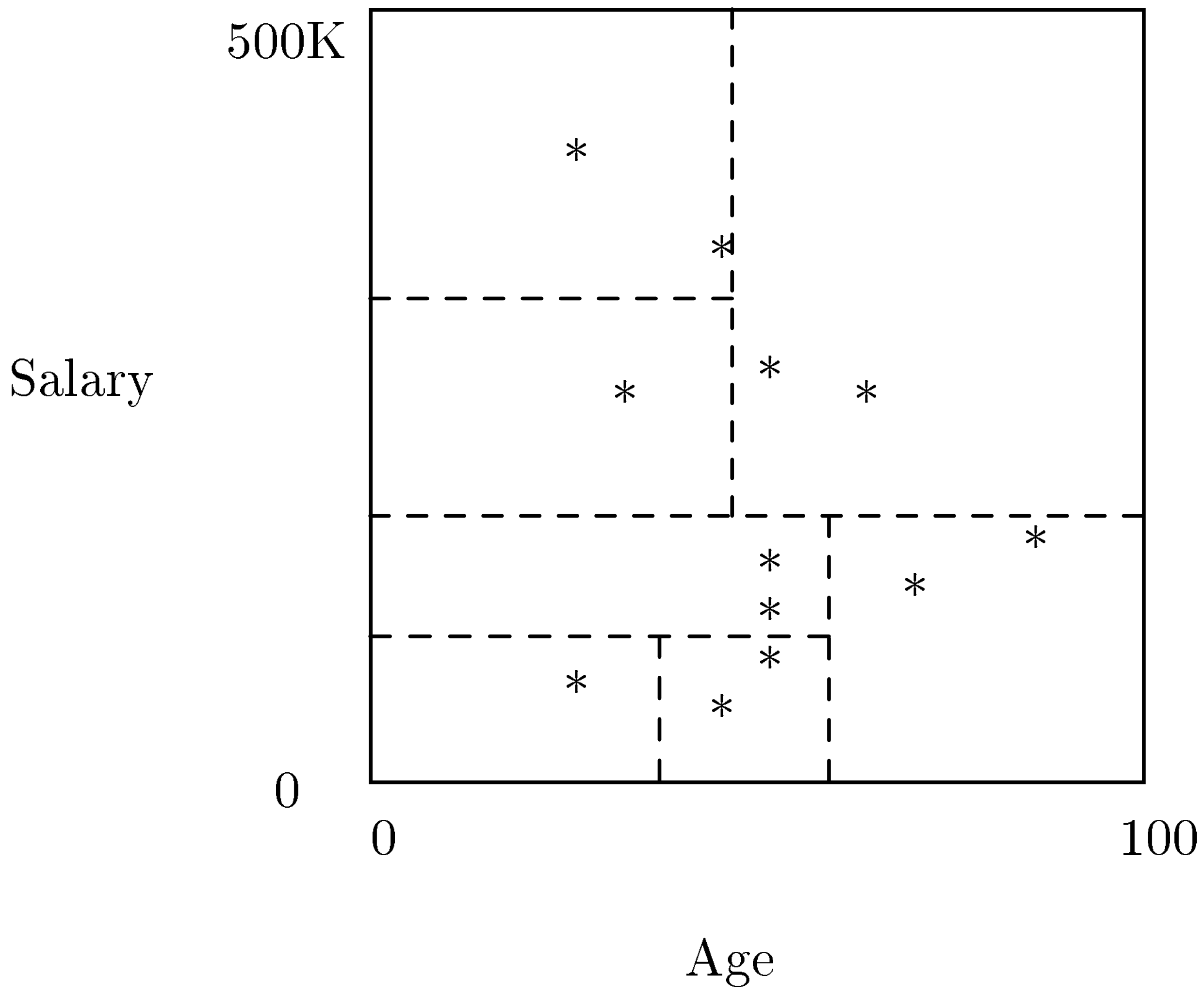
60
400
260
60
350
75
100
120
275
260
110
140
25
30
45
50
60
70
85

KD-Trees

- A main-memory data structure based on binary search trees.
  - Can be fixed somewhat to adapt to the block model of storage.
- Levels rotate among the dimensions, partitioning the points by comparison with a value for that dimension.
  - Real kd-trees keep a point at each node; we shall not.

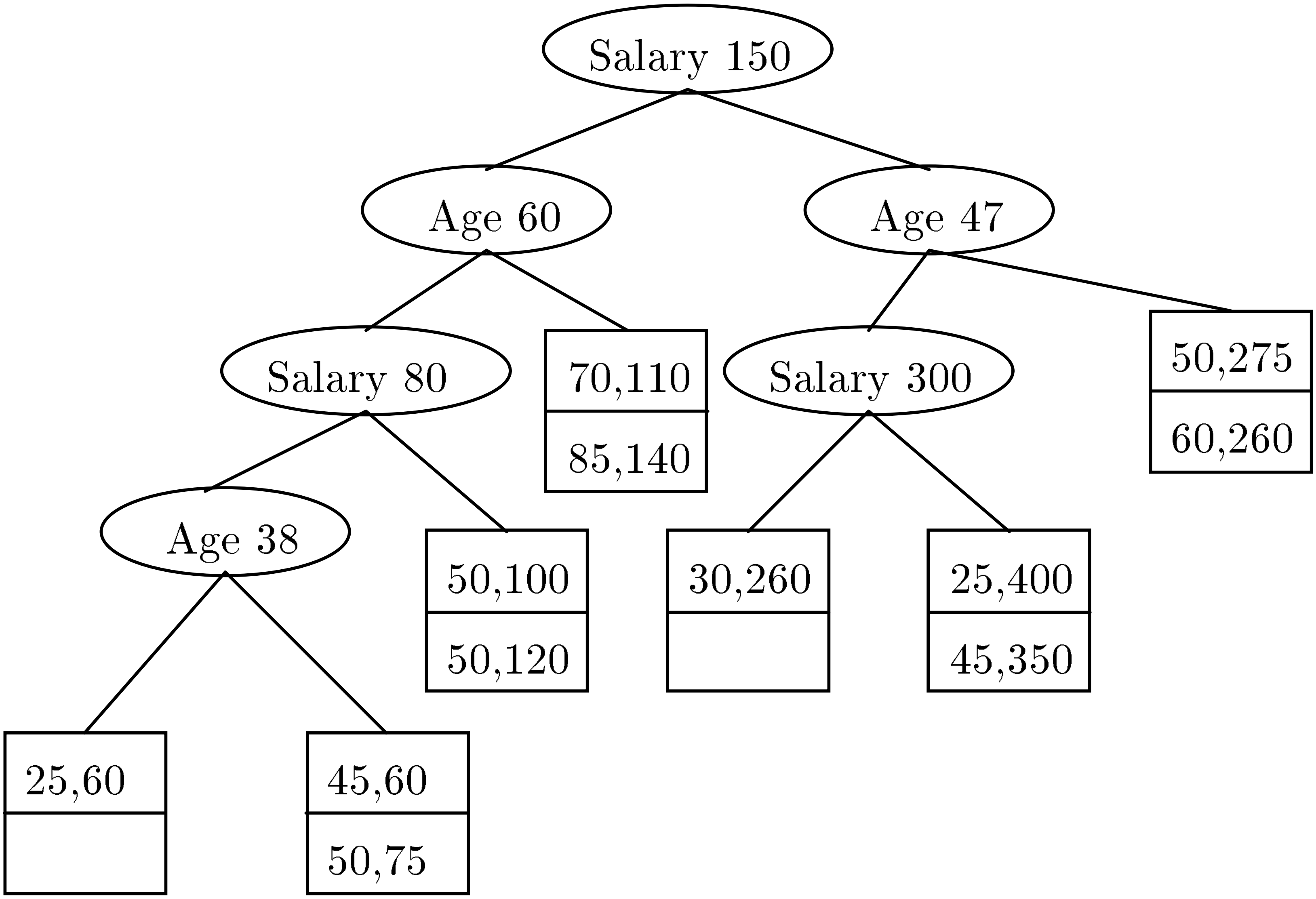
Example

Gold-jewelry data:



KD-Tree Following This Recursive Partition

Assume records are in blocks at leaves, two to a block.



Adapting to Block Model

1. Tree nodes split many ways, like B-trees.

   ❖ Risk: too many null pointers or data partitioned too finely at bottom.

   ❖ Partial solution: leaves can share blocks.

2. Pack many interior nodes (forming a subtree) into one block.

   ❖ Risk: it may not be possible to group nodes at lower levels productively.

Lookup in KD-Trees

Find appropriate leaf by binary search. Is the record there?

Insert Into KD-Trees

- Lookup record to be inserted, reaching the appropriate leaf.
- If there is room, put record in that block.
- If not, find a suitable value for the appropriate dimension and split the leaf block.

Example

Someone 35 years old with a salary of \$500K buys gold jewelry.

- Belongs in leaf with $(25, 400)$ and $(45, 350)$.
- Too full: split on age.