CS525: Advanced Database Organization

Notes 5: Indexing and Hashing
Part IV: Indexes and More

Yousef M. Elmehdwi

Department of Computer Science

Illinois Institute of Technology

yelmehdwi@iit.edu

October 25th 2023

Slides originally prepared/adapted from courses taught by Andy Pavlo, Carnegie Mellon University & Shun Yan Cheung, Emory University

Topics

- Additional Index Usage
- Tries/Radix Trees
- Inverted Indexes
- Multi-dimensional index structures
- Bitmap Indexes

Additional Index Usage

- Indices are a way to provide fast access to data items.
- There are a number of indexing methods and tools that databases can use to improve performance.
  - Implicit Indexes
  - Partial Indexes
  - Covering Indexes
  - Index Include Columns
  - Function/Expression Indexes

Implicit Indexes

- These are indexes created automatically by the DBMS for primary keys and unique constraints.
- They ensure data integrity and enforce data uniqueness.
- Most DBMSs automatically create an index to enforce integrity constraints but not referential constraints (foreign keys).

```
CREATE TABLE foo (
  id SERIAL PRIMARY KEY,
  val1 INT NOT NULL,
  val2 VARCHAR(32) UNIQUE
);
```

```
CREATE UNIQUE INDEX foo_pkey ON foo(id);
```

```
CREATE UNIQUE INDEX foo_val2_key ON foo(val2);
```

Implicit Indexes

- Most DBMSs automatically create an implicit index to enforce integrity constraints but not referential constraints (foreign keys).
  - Primary Keys
  - Unique Constraints

```
CREATE TABLE foo (
  id SERIAL PRIMARY KEY,
  val1 INT NOT NULL,
  val2 VARCHAR(32) UNIQUE
);
```

```
CREATE TABLE bar (
  id INT REFERENCES foo(val1),
  val VARCHAR(32)
);
```

- Not working, unless we create an index:

```
CREATE INDEX foo_val1_key ON foo(val1);
```

Implicit Indexes

- Most DBMSs automatically create an index to enforce integrity constraints but not referential constraints (foreign keys).
  - Primary Keys
  - Unique Constraints
- or

```
CREATE TABLE foo (
  id SERIAL PRIMARY KEY,
  val1 INT NOT NULL UNIQUE,
  val2 VARCHAR(32) UNIQUE
);
```

```
CREATE TABLE bar (
  id INT REFERENCES foo(val1),
  val VARCHAR(32)
);
```

Partial Indexes[1]

- In many situations, a user may not need an index for every tuple in the table and may only be interested in a subset of the data.
- A partial index is a type of index that includes only a subset of the rows in a table based on a specified condition (WHERE clause).
- Create an index on a subset of the entire table.
- It's used to index a specific subset of data that's frequently queried, which can save space and improve query performance for that subset.
- Partial indexes are especially useful when dealing with large tables where not all data needs to be indexed.

[1] https://www.postgresql.org/docs/current/indexes-partial.html

Partial Indexes[1]

```
CREATE INDEX idx_foo
    ON foo (a, b)
    WHERE c = 'Alice';
```

```
SELECT b
FROM foo
WHERE a = 123 AND c = 'Alice';
```

- One common use case is to partition indexes by date ranges.
  - Create a separate index per month, year.

[1] https://www.postgresql.org/docs/current/indexes-partial.html

Covering Indexes[2]

- A covering index is an index that contains all the columns needed to satisfy a query, eliminating the need to access the underlying table.
- This can significantly improve query performance by reducing I/O operations.
- Covering indexes are designed to "cover" a query's needs, making it unnecessary to access the actual data rows.
- The DBMS can complete the entire query just based on the data available in the index.

```sql
CREATE INDEX idx_foo ON foo(a,b);
```

```sql
SELECT b
FROM foo
WHERE a = 123;
```

[2] https://www.postgresql.org/docs/current/indexes-index-only-scans.html

Index Include Columns[3]

- Index include columns allow users to include additional non-key columns along with the indexed columns.
  - These extra columns are only stored in the leaf nodes and are not part of the search key.
- This can improve query performance by making it possible to retrieve required data directly from the index without accessing the table itself.

```
CREATE INDEX idx_foo
ON foo(a, b)
INCLUDE (c)
```

- INCLUDE clause specifies a list of columns which will be included in the index as non-key columns
  - A non-key column cannot be used in an index scan search qualification, and it is disregarded for purposes of any uniqueness or exclusion constraint enforced by the index.

```
SELECT b FROM foo WHERE a = 123 AND c = 'Alice';
```

[3] https://www.postgresql.org/docs/current/sql-createindex.html

Functional/Expression Indexes

- In some cases, we might need to create an index on a computed or derived value (e.g., concatenation of columns, mathematical operations, or date transformations).
- fFunction/expression indexes allow us to index the result of such operations, making it more efficient to query based on these computed values.
- An index does not need to store keys in the same way that they appear in their base table.
- Instead, function/expression indexes store the output of a function or expression as the key instead of the original value.
  - *An index column need not be just a column of the underlying table, but can be a function or scalar expression computed from one or more columns of the table. This feature is useful to obtain fast access to tables based on the results of computations.*[4]
- It is the DBMS's job to recognize which queries can use that index.

[4] https://www.postgresql.org/docs/current/indexes-expressional.html

Functional/Expression Indexes

- An index does not need to store keys in the same way that they appear in their base table.

```
SELECT * FROM users
WHERE EXTRACT(dow FROM login) = 2;
```

```
-- Not Working
CREATE  INDEX idx_user_login ON users (login);
```

- You can use expressions when declaring an index.

```
CREATE INDEX idx_user_login
ON users (EXTRACT(dow FROM login));
```

- (using partial index)

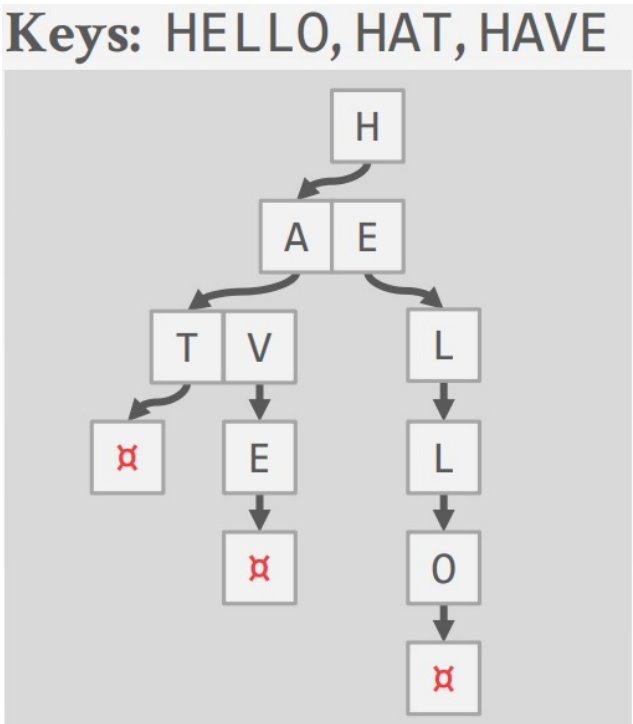
```
CREATE INDEX idx_user_login
ON foo (login)
WHERE EXTRACT(dow FROM login) = 2;
```

Observation

- The inner node keys in a `B+-Tree` cannot tell you whether a key exists in the index. You must always traverse to the leaf node.
- This means that you could have (at least) one buffer pool page miss per level in the tree just to find out a key does not exist.

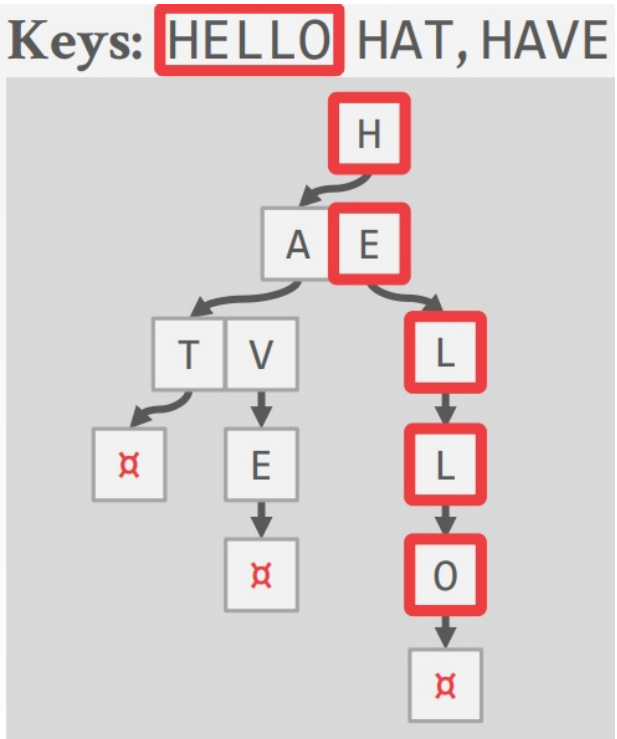
Trie Index

- A trie is a tree data structure that is commonly used to store and search for strings or keys that can be represented as sequences of characters.
- A trie tree, the branching at any level is determined by only a digit of the key.
  - Digits: Subset of a key.
- It uses digital representation of keys to examine prefixes one-by-one instead of comparing entire key.
  - Also known as Digital Search Tree, Prefix Tree.



Trie Index

- Use a digital representation of keys to examine prefixes one-by-one instead of comparing entire key.
  - Also known as Digital Search Tree, Prefix Tree.

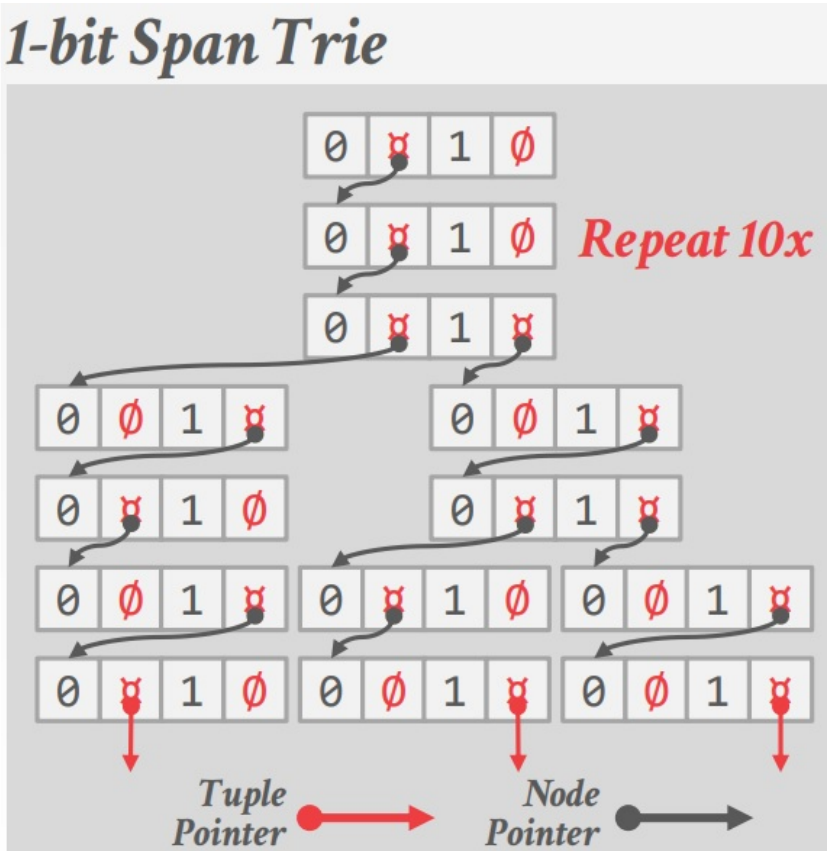


Trie Index Properties

- Shape only depends on key space and lengths.
  - Does not depend on existing keys or insertion order.
  - Does not require rebalancing operations.
- All operations have $O(k)$ complexity where k is the length of the key.
  - The path to a leaf node represents the key of the leaf
  - Keys are stored implicitly and can be reconstructed from paths.

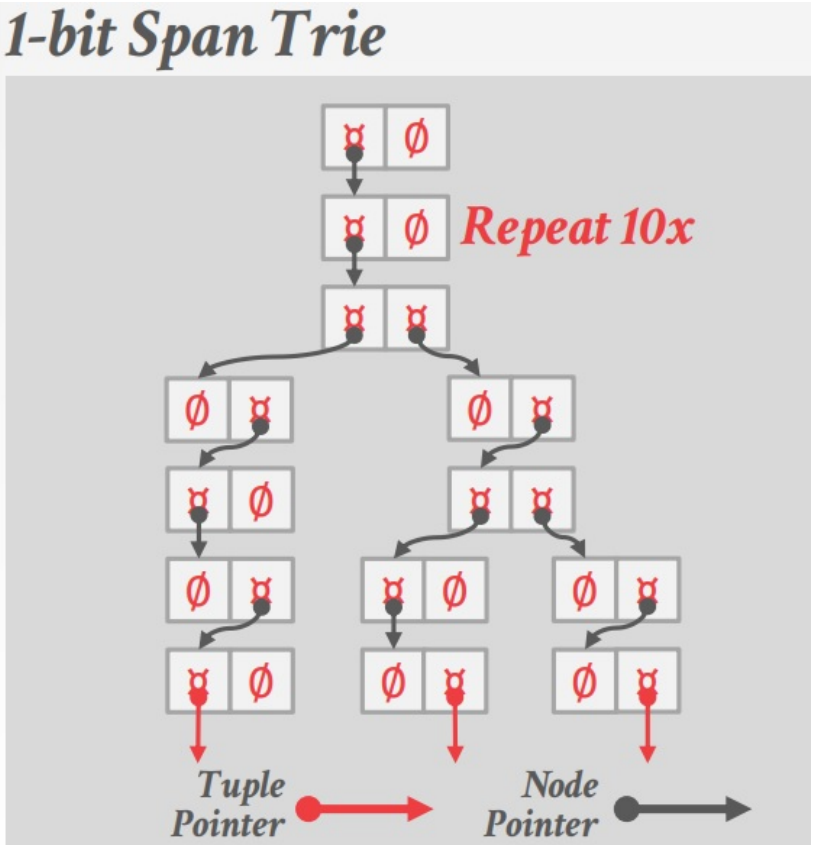
Trie Index

```
K10 → 00000000 00001010
K25 → 00000000 00011001
K31 → 00000000 00011111
```



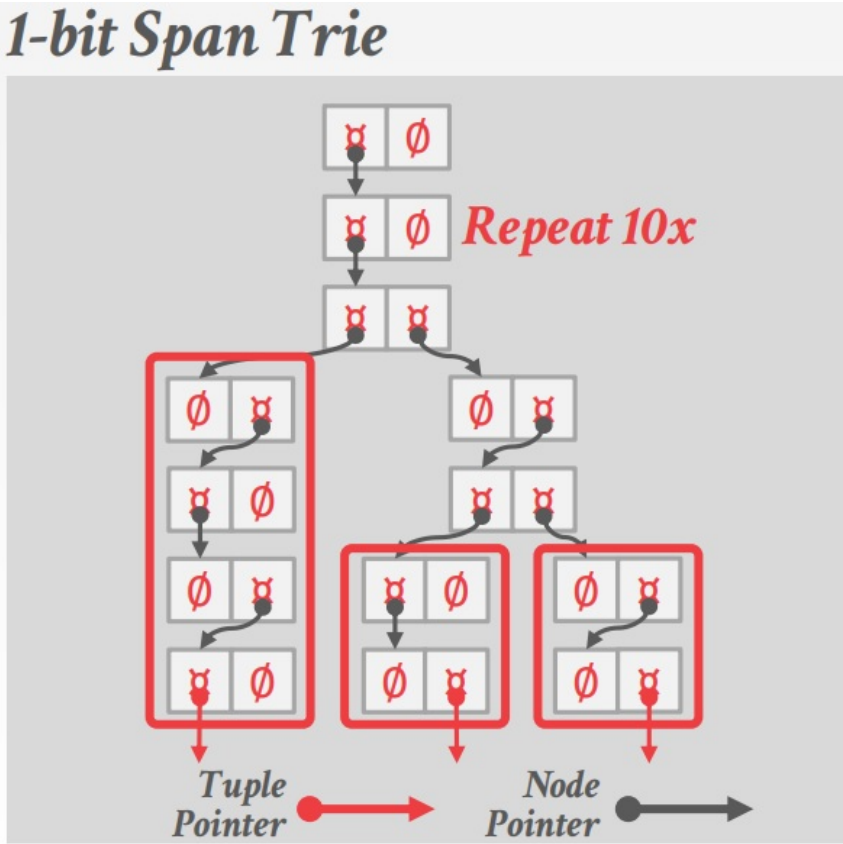
Trie Index

```
K10 → 00000000 00001010
K25 → 00000000 00011001
K31 → 00000000 00011111
```



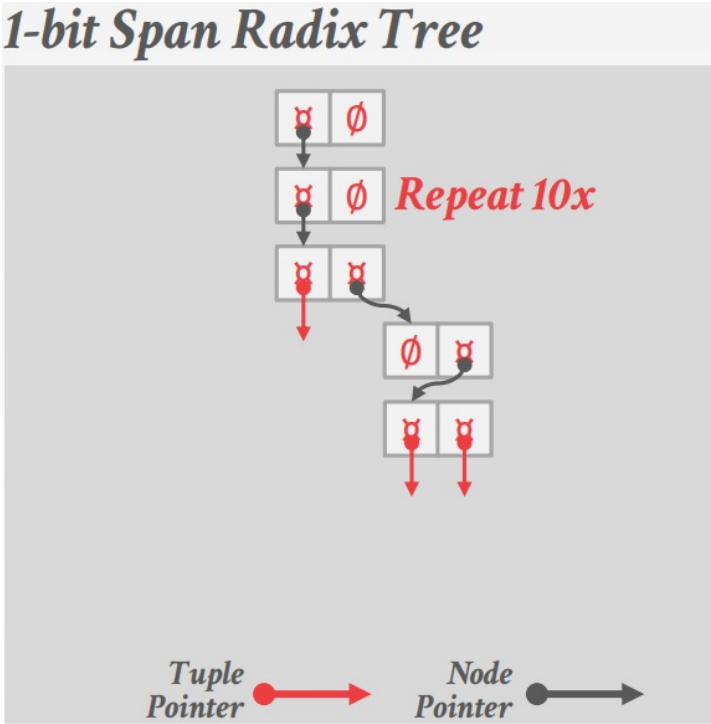
Trie Index

```
K10 → 00000000 00001010
K25 → 00000000 00011001
K31 → 00000000 00011111
```



Radix Tree

```
K10 → 00000000 00001010
K25 → 00000000 00011001
K31 → 00000000 00011111
```

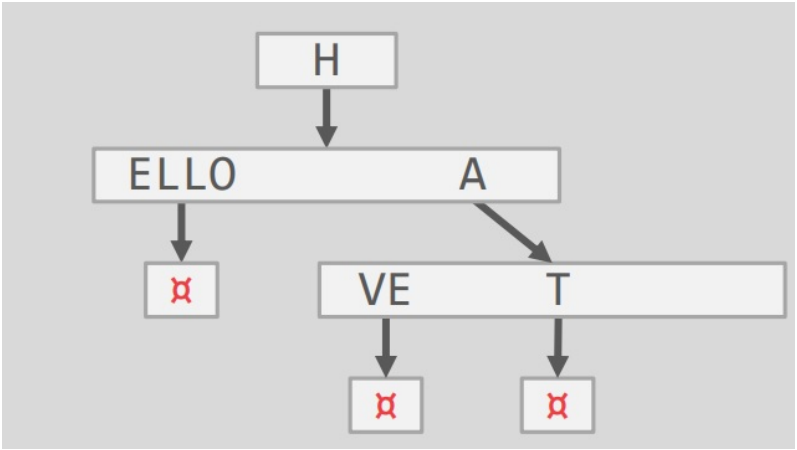


- Omit all nodes with only a single child.
- Can produce false positives, so the DBMS always checks the original tuple to see whether a key matches.

Radix tree

- A radix tree is a variant of a trie data structure. It uses digital representation of keys to examine prefixes one-by-one instead of comparing entire key.
- It is different than a trie in that there is not a node for each element in key, nodes are consolidated to represent the largest prefix before keys differ.
- The height of tree depends on the length of keys and not the number of keys like in a `B+-Tree`.
- The path to a leaf nodes represents the key of the leaf. Not all attribute types can be decomposed into binary comparable digits for a radix tree.

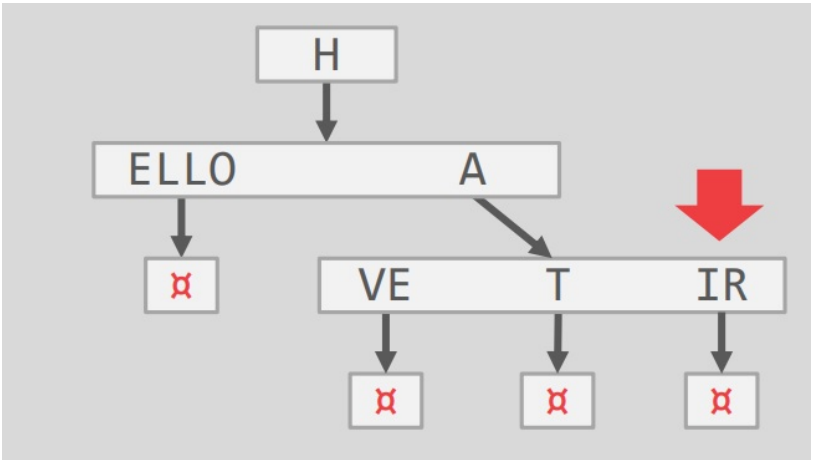
Radix Tree: Modification




```
Insert HAIR
Delete HAT, HAVE
```

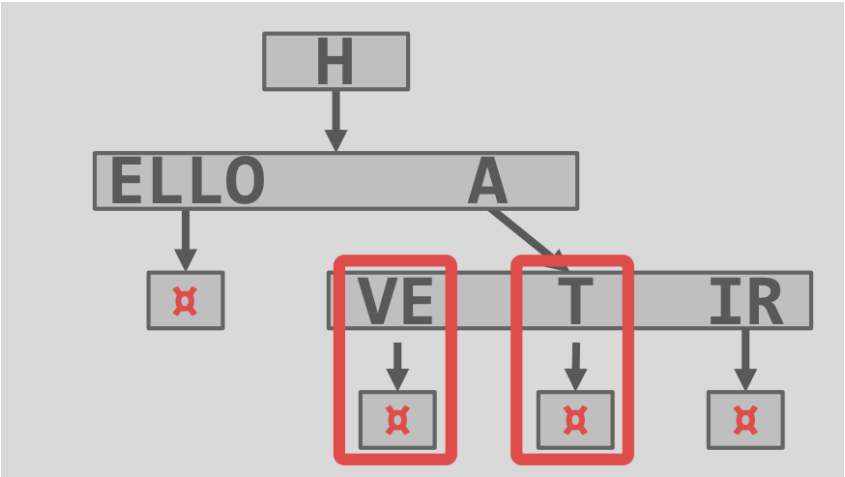
Radix Tree: Modification

```
Insert HAIR
Delete HAT, HAVE
```



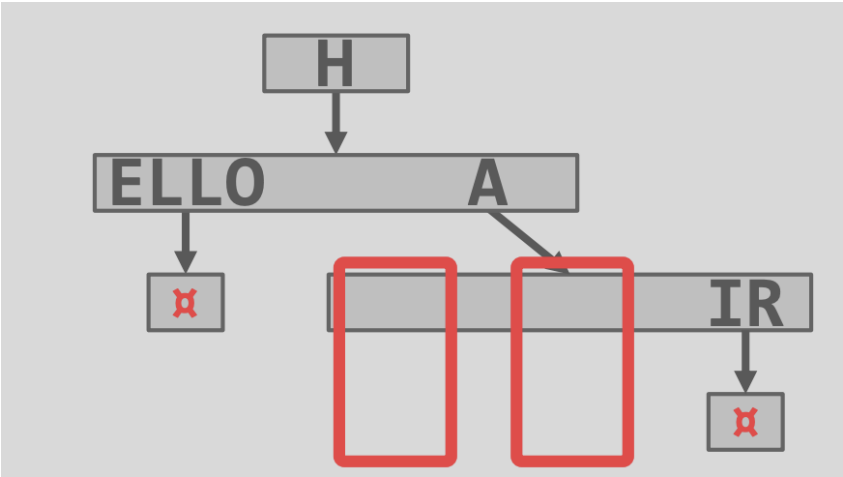
Radix Tree: Modification

```
Insert HAIR
Delete HAT, HAVE
```



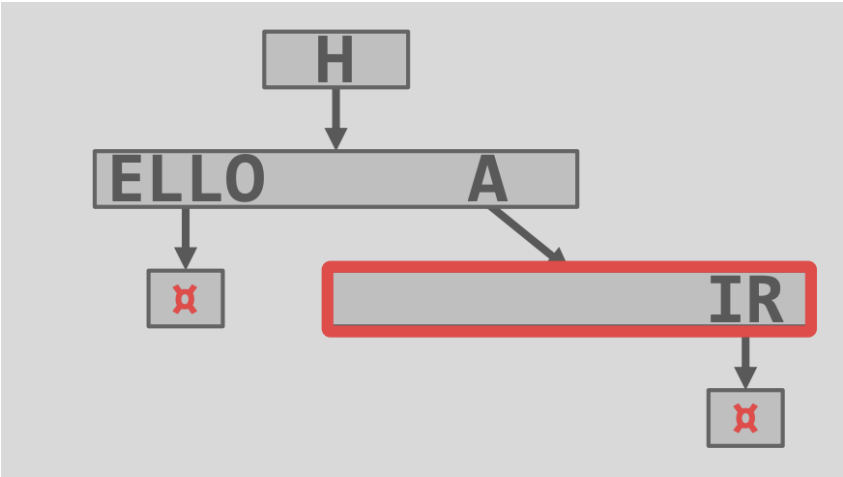
Radix Tree: Modification

```
Insert HAIR
Delete HAT, HAVE
```



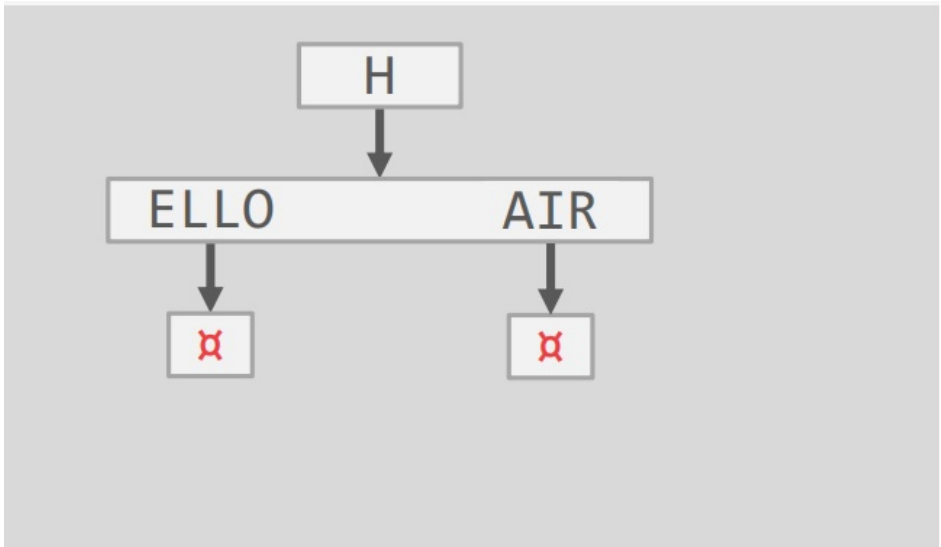
Radix Tree: Modification

```
Insert HAIR
Delete HAT, HAVE
```



Radix Tree: Modification

```
Insert HAIR
Delete HAT, HAVE
```



Observation

- The tree indexes that we've discussed so far are useful for `point` and `range` queries:

```
- Find all customers in the 60616 zip code.
- Find all orders between June 2018 and September 2018.
```

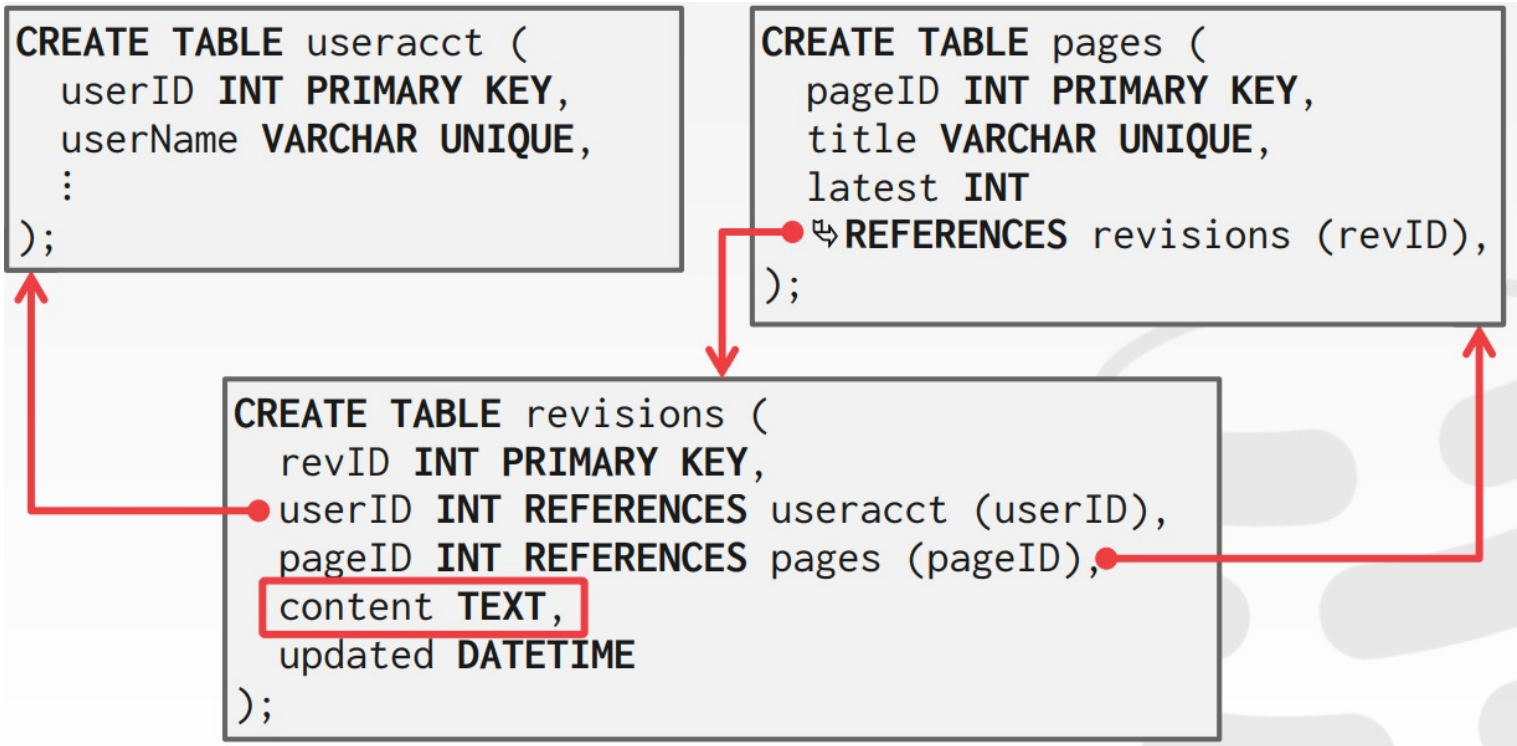
- They are not good at keyword searches
  - searching for documents containing specific keywords, e.g., web search engines

```
- Find all Wikipedia articles that contain the word "Database"
```

- How can one make such indexes?

- *Tee-based indexes, such as B^+-Trees and Radix Trees, are indeed well-suited for point and range queries, as they efficiently support operations involving comparisons on keys or ranges of keys. However, they are not the ideal choice for full-text search or keyword searches, where you want to find documents containing specific words or phrases.*

WIKIPEDIA Example



WIKIPEDIA Example

- If we create an index on the *content* attribute, what does that do?

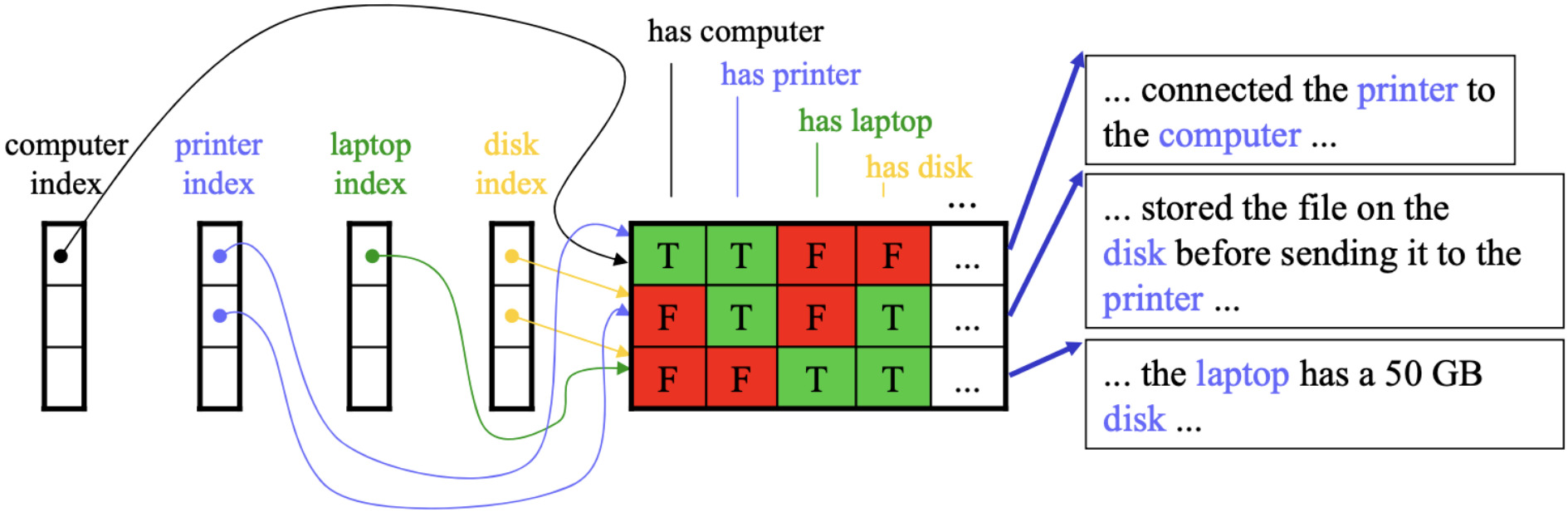
```
CREATE INDEX idx_rev_cntnt ON revisions (content);
```

- This doesn't help our query. Our SQL is also not correct.

```
SELECT pageID FROM revisions
WHERE content LIKE '%Database%';
```

Inverted Indexes

- Approach 1: Boolean Inverted Index (true/false table)
  - define "all" keywords
  - make table with one boolean attribute for each keyword
  - make one tuple per document/text
  - make index on all attributes including only `TRUE` values
- Example: allowed keywords: computer, printer, laptop, disk,...

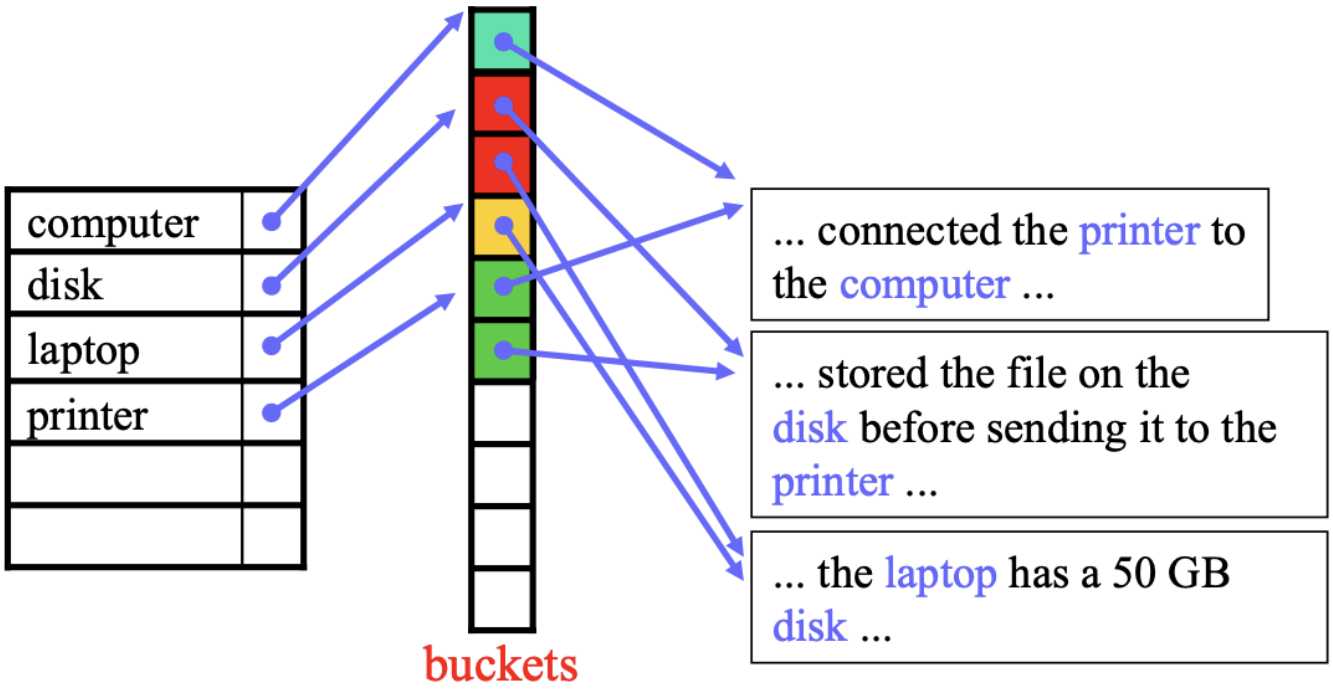


Inverted Indexes

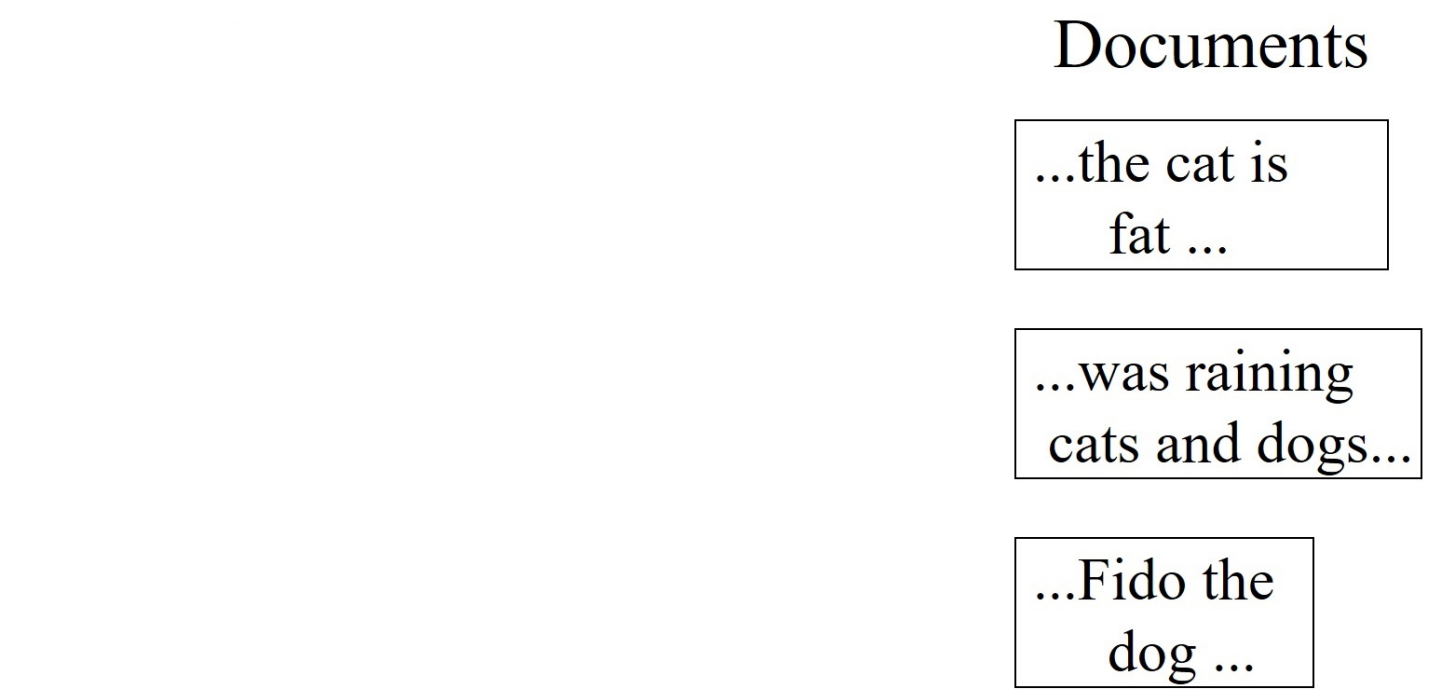
- An inverted index stores a mapping of words to records that contain those words in the target attribute.
  - Sometimes called a full-text search index.
- Most of the major DBMSs support inverted indexes natively, but there are specialized DBMSs where this is the only table index data structure available.
- Inverted indexes enable various text search features, including keyword searches, phrase searches, fuzzy searches (for approximate matches), and relevance ranking of search results.
- The creation of an inverted index typically involves tokenization (breaking text into individual words or terms) and stemming (reducing words to their root forms).
- These processes improve the accuracy of text-based searches.

Inverted Indexes

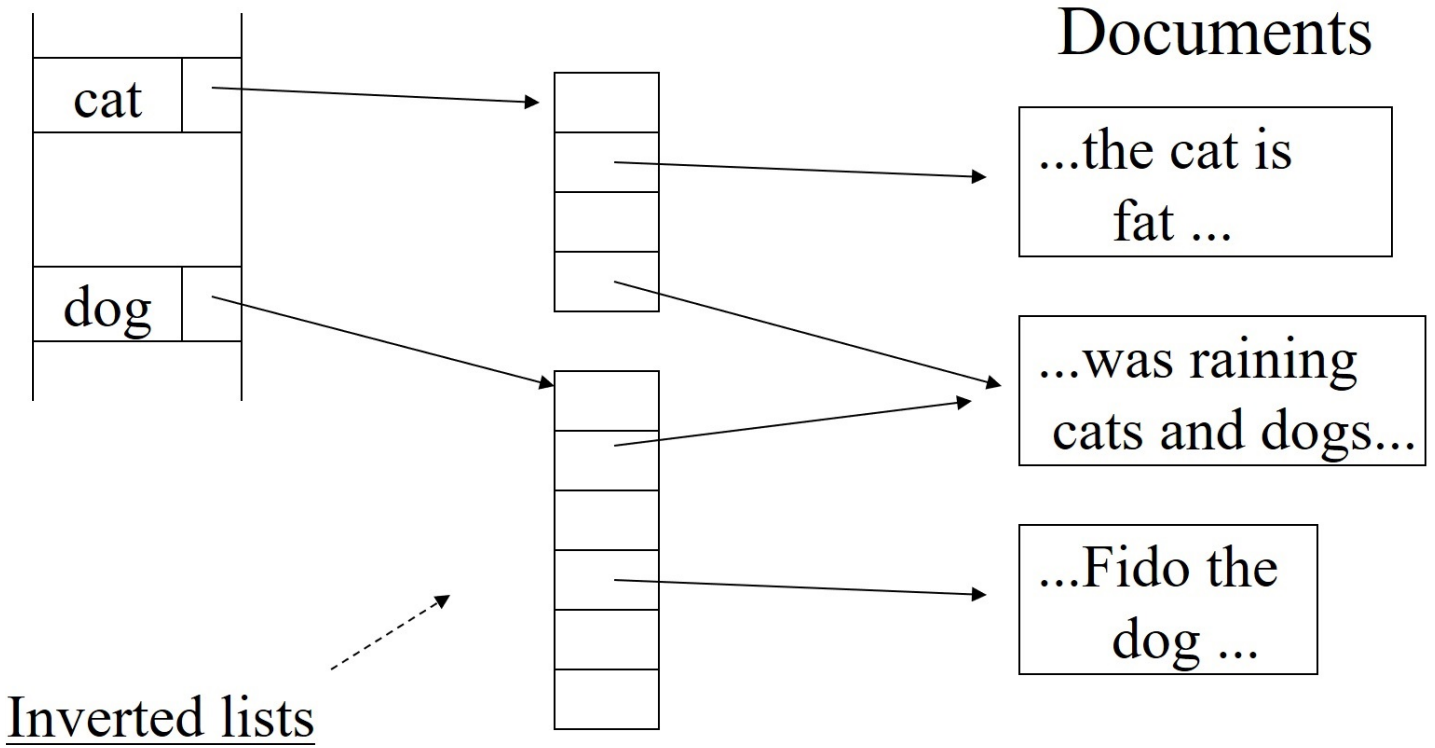
- Approach 2: inverted index:
  - make one (inverted) index for keywords
  - use indirection putting pointers in buckets
- Example



This idea used in text information retrieval



This idea used in text information retrieval



`inverted index` consists of a set of word-pointer pairs; the words are in effect the search key for the index.

Query Types

Inverted indexes allow users to do three types of queries that cannot be performed on `B+Trees`:

- Phrase Searches
  - Find records that contain a list of words in the given order.
- Proximity Searches
  - Find records where two words occur within n words of each other.
- Wildcard Searches
  - Find records that contain words that match some pattern (e.g., regular expression).

Query Types

- Find articles with “cat” and “dog”
- Find articles with “cat” or “dog”
- Find articles with “cat” and not “dog”
- Find articles with “cat” in title
- Find articles with “cat” and “dog” within 5 words

Design Decisions

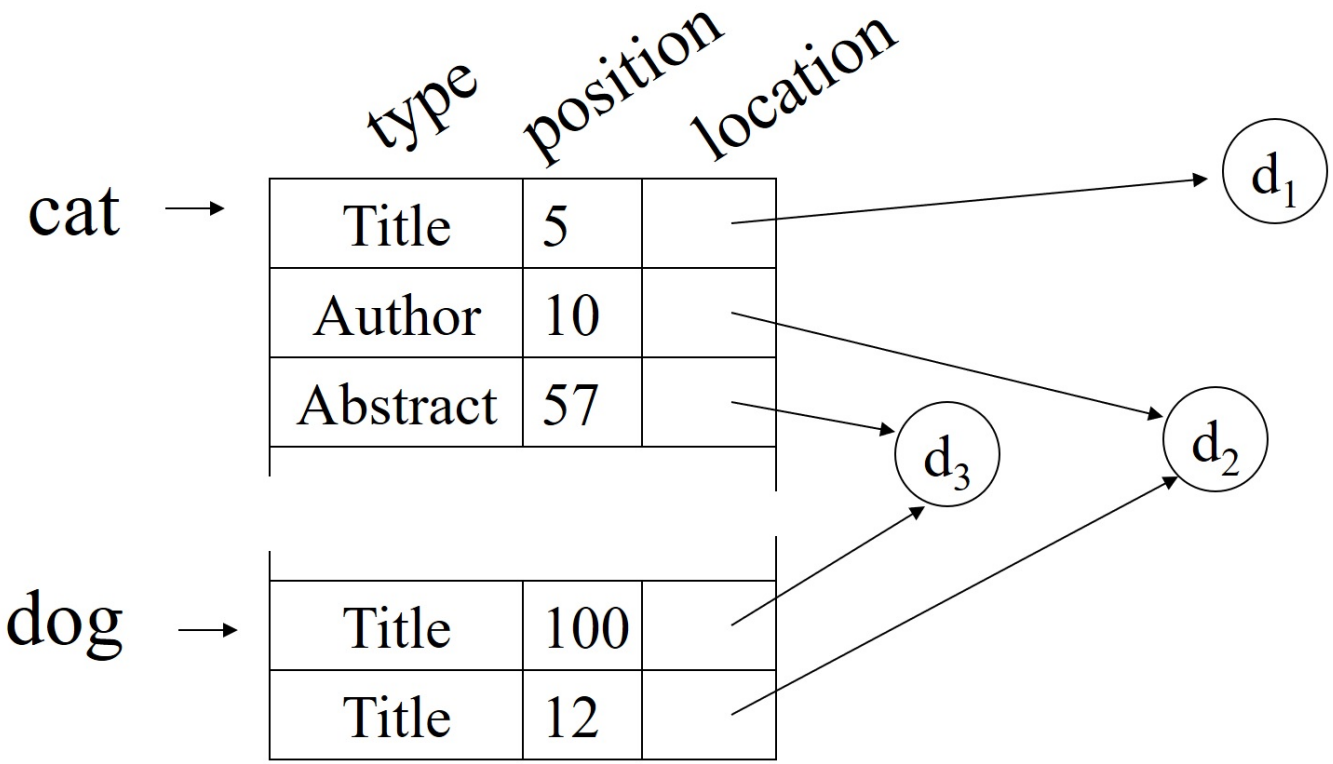
The two primary decision decisions for developing inverted indexes are what to store and when to update.

- Decision #1: What To Store
  - The index needs to store at least the words contained in each record (separated by punctuation characters).
  - Can also store additional information such as the word frequency, position, and other meta-data (such as creation date, author, or document category).
  - The specific data stored depends on the requirements of the application, and it may vary from system to system.
- Decision #2: When To Update
  - Updating an inverted index every time the table is modified is expensive and slow.
  - Thus, most DBMSs will maintain auxiliary data structures to stage updates and then update the index in batches.
    - During this batch indexing process, the changes recorded in the auxiliary data structures are incorporated into the index.
- Reading: Generalized Inverted Index, PostgreSQL[5]

[5] "*GIN stands for Generalized Inverted Index. GIN is designed for handling cases where the items to be indexed are composite values, and the queries to be handled by the index need to search for element values that appear within the composite items. For example, the items could be documents, and the queries could be searches for documents containing specific words.*"

Common technique: more info in inverted list

- Can also store frequency, position, and other meta-data.



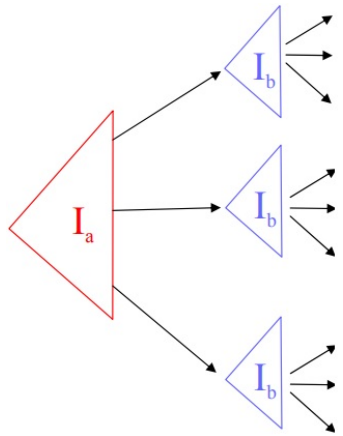
Recap

- We have studied the following 3 index structures:
  - Conventional indexes
  - `B+-tree` indexes
  - `Hash-based` indexes
- Common property
  - The `search key` values are values taken from a `one-dimensional` space/set
  - Example
    - If we index the search keys that are names, the search key values can be ordered on a one-dimensional space: Search keys:

```
AAAA < AAAB < ... < ZZZZ
```

Recall: Using Indexes in Queries: Multiple Key index

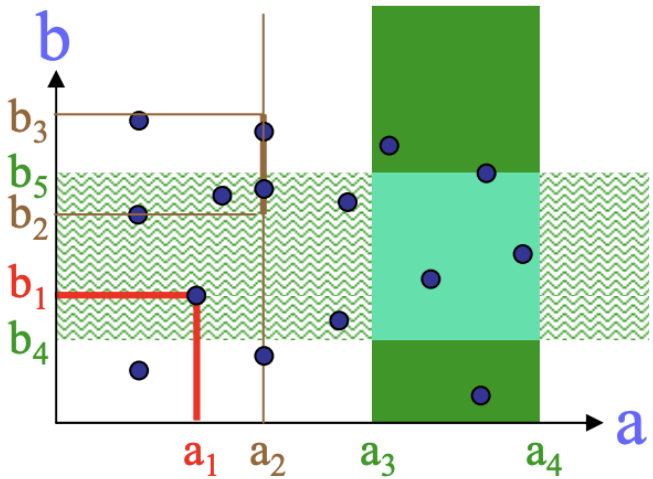
- For which queries is this index good?



  - ☺ find records where `a = 10 AND b = 'x'`
  - ☺ find records where `a = 10 AND b` $\geq$ `'x'`
  - ☺ find records where `a = 10`
  - ☹ find records where `b = 'x'`
  - ? find records where `a` $\geq$ `10 AND b = 'x'`
    - may search several indexes in next dimension
    - better if dimensions changed order
- Several other approaches
  - other tree-like structures
  - hash-like structures
  - (bitmap-indexes)

Map View

- A multidimensional index is a data structure that can handle data with multiple attributes or dimensions, such as spatial coordinates (e.g., latitude and longitude for geographic data), properties of multi-attribute records, or any other dataset with two or more characteristics.
- A multidimensional index combines several dimensions into one index
- If we assume only two, we can imagine that our index is a geographical map:



- Now our search is similar to searching the map for:
  - points: a_1 and b_1
  - lines: a_2 and $<b_2, b_3>$
  - areas: $<a_3, a_4>$ and $<b_4, b_5>$

Multi-dimensional Information

- There are information that are naturally `multi-dimensional`
  - e.g., `Geographic information:`
    - Stores objects in a (typically) two-dimensional space.
    - The objects may be points or shapes.
    - Often, these databases are maps, where the stored objects could represent houses, roads, bridges, pipelines, and many other physical objects.

Spatial Data

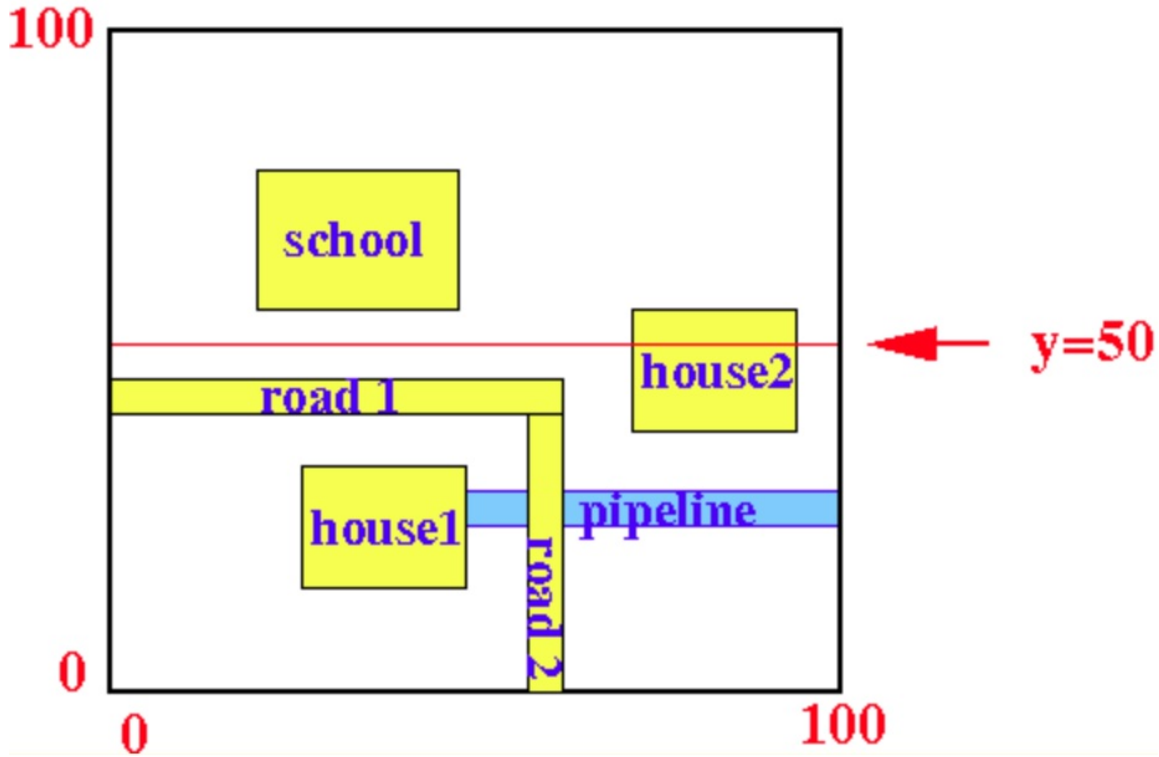
- Spatial data refers to data referring to a point or a region in two or higher dimensional space.
- Spatial databases are capable of storing various data types, including lines, polygons, and more.
  - Allows relational databases to store and retrieve spatial information
  - Queries can use spatial conditions (e.g. contains or overlaps).
  - Queries can mix spatial and non-spatial conditions

Querying multi-dimensional data

- Partial Match queries
- Range queries
- Nearest neighbor queries
- Where-am-I queries

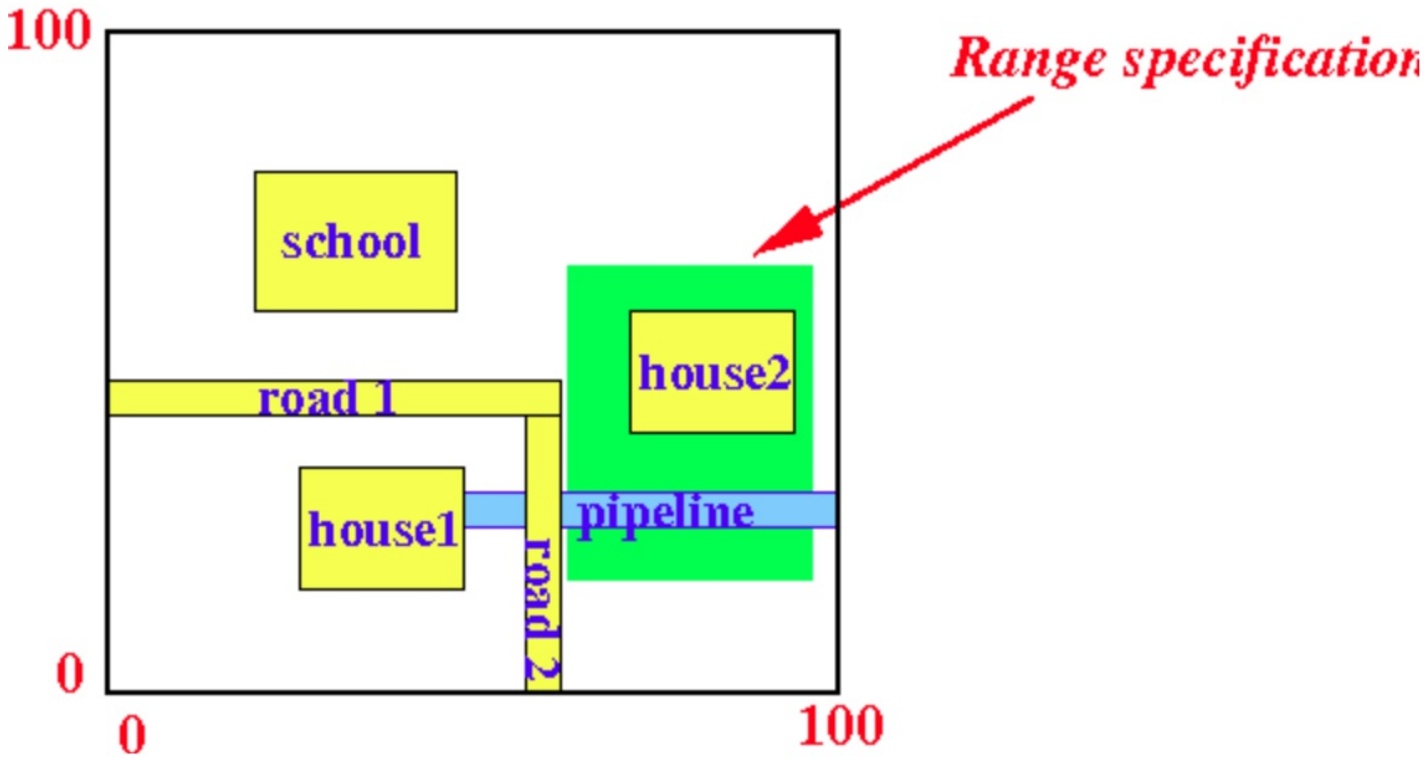
Partial Match queries

- The query specifies conditions on some dimensions but not on all dimensions
- e.g., find all points/objects that intersects with $y = 50$



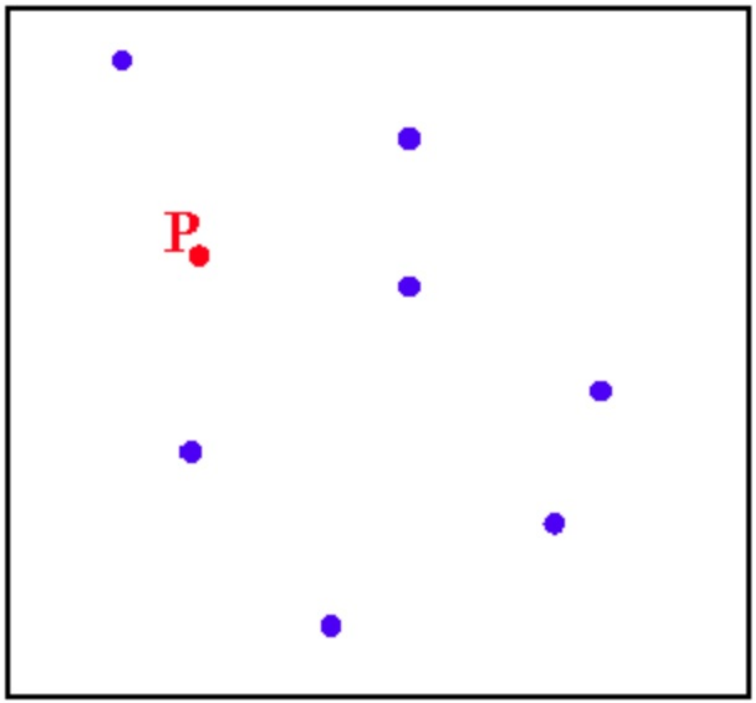
Range queries

- Deal with spatial regions.
- Find objects that are located either partially or wholly within a certain range
  - ask for objects that lie partially or fully inside a specified region.
- e.g., find all objects that have an overlap with the green area:



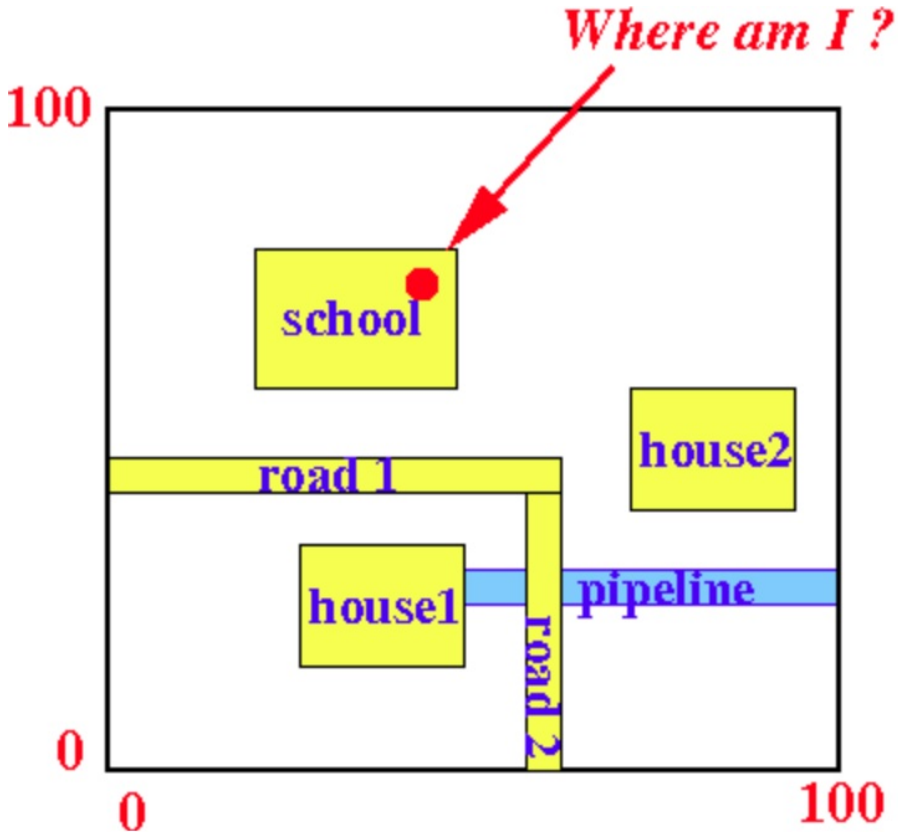
Nearest neighbor queries

- Find the closest point to a given point.
  - given a point or an object, find the nearest object that satisfies given conditions.
- Suppose we have a relation containing points on a map
- Each point is stored in the following relation as `Point(x,y)`
- Find the point that is closest to point `P(10,20)`



Where-am-I queries

- Given a location (i.e., coordinate)
- Find the object(s) that contains the location



Multi-dimensional indexes

- Are muliti-dimensional indexes necessary?
- Can `one-dimensional index` technique support multi-dimensional queries efficiently?

Multi-dimensional indexes

- We cannot store `geographically` "`related`" `data` `randomly`
  - If related `geographical data` is store `randomly`, we will need to access too many `data blocks`
- $\Rightarrow$ must store `geographically` "`related`" `data` (i.e.: points that are close to each other) in the same `data block`
- To support the access to such a data
  - Need a more appropriate `index structure` for `multi-dimensional data`

Multi-dimensional index structures

- Multi-dimensional index: an index on the multi-dimensional data that help us answer a (commonly used) multi-dimensional query (more) efficiently.
- Most data structures for supporting queries on multi-dimensional data fall into one of two categories:

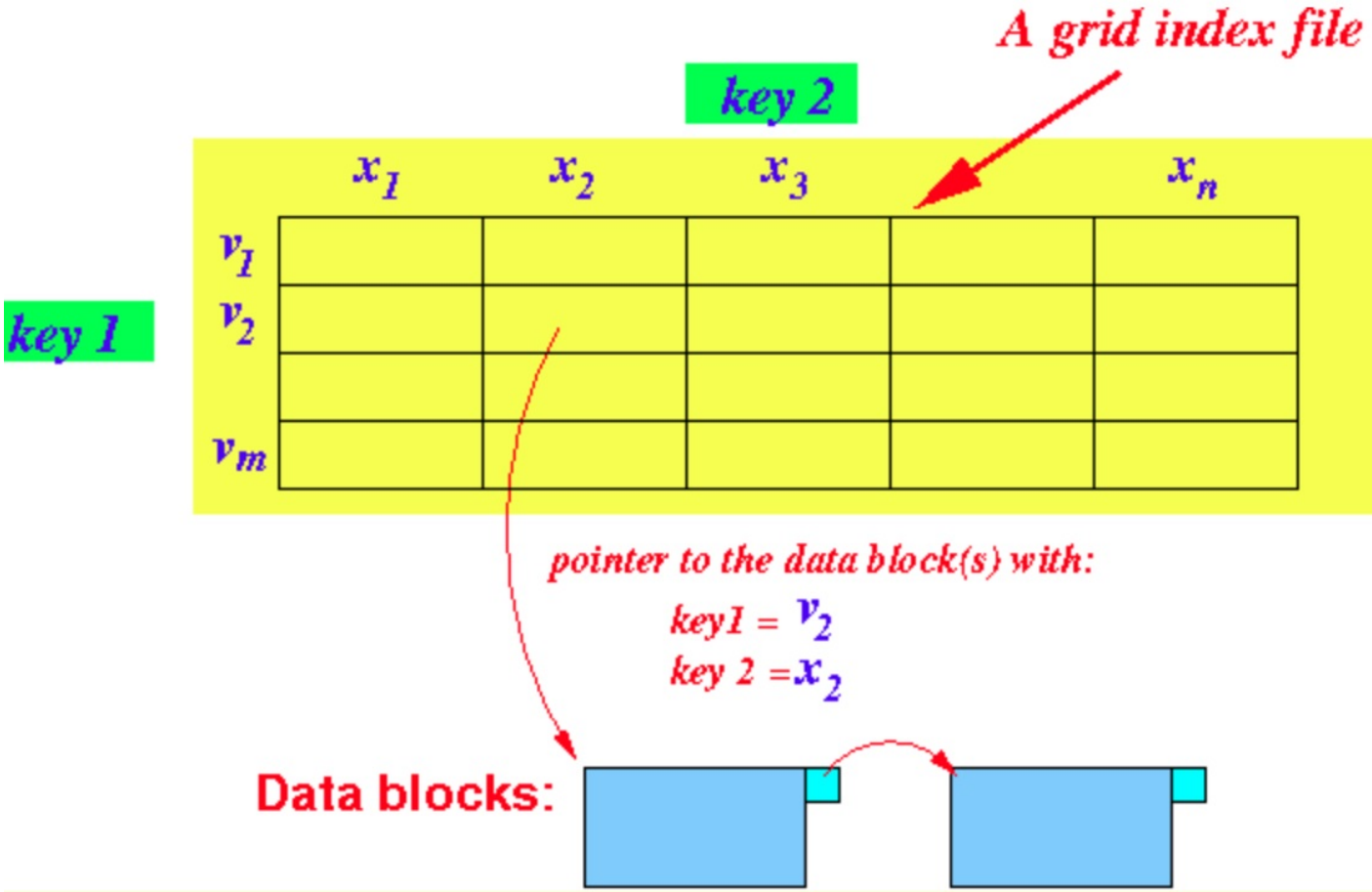
1. Table-based (Hash like) structures
   - Grid Index files
   - Partitioned Hashing functions
2. Tree like (Tree based) structures
   - Multiple key indexes
   - kd-trees
   - Quad trees
   - R-trees

Grid Index

- Partition multi-dimensional space with a grid
- Grid files extend traditional hashing indexes
  - hash values for each attribute in a multidimensional index
  - usually does not hash values, but regions `h(key) = <x,y>`
  - grid lines partition the space into stripes
  - Intersections of stripes from different dimensions define regions
- The number of grid lines in different dimensions may vary.
- Spacings between adjacent grid lines may also vary.
- Each region corresponds to a bucket.
- Attribute values for record determine region and therefore bucket

Grid Index

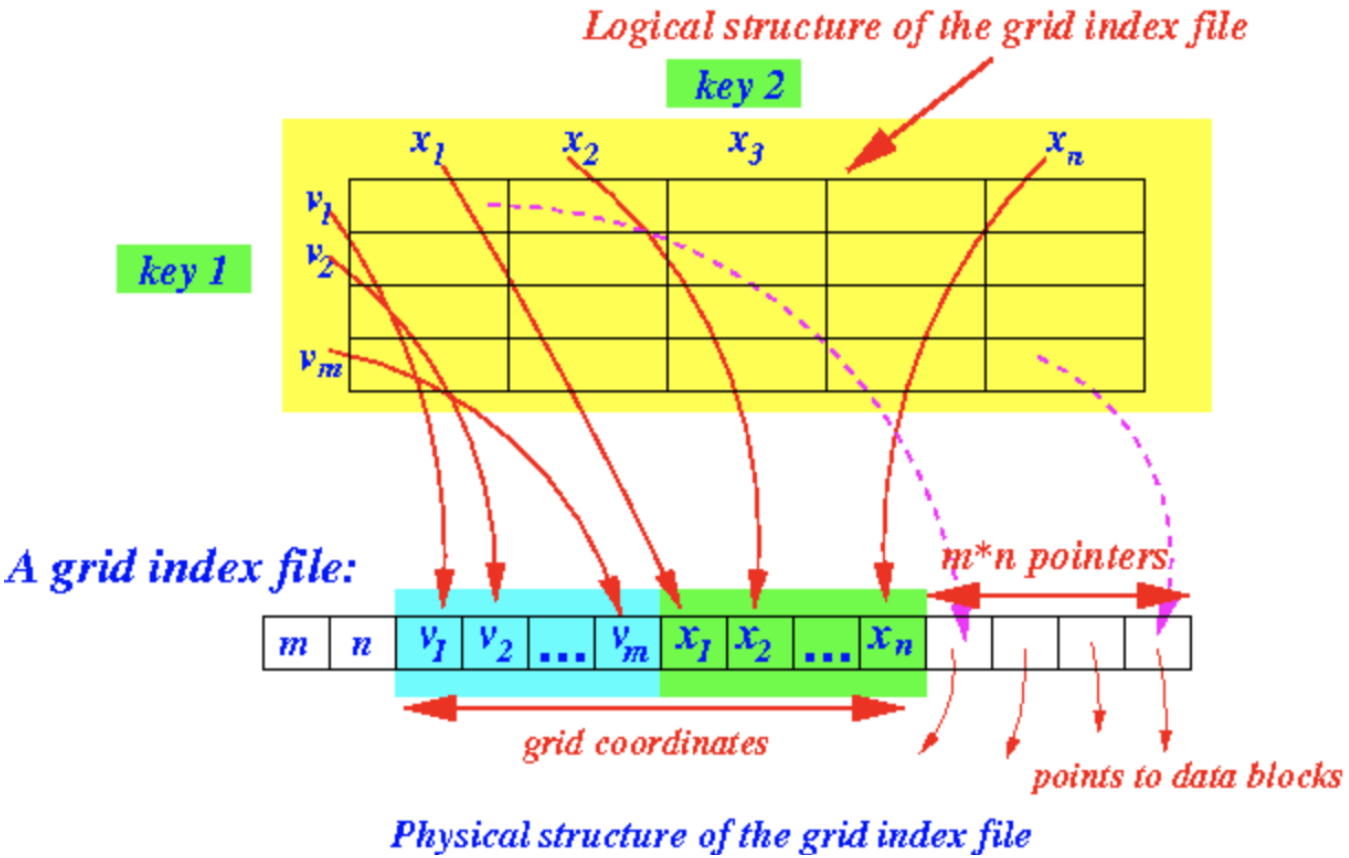
- `Grid index file:` an index that is organized into a 2-dimensional structure



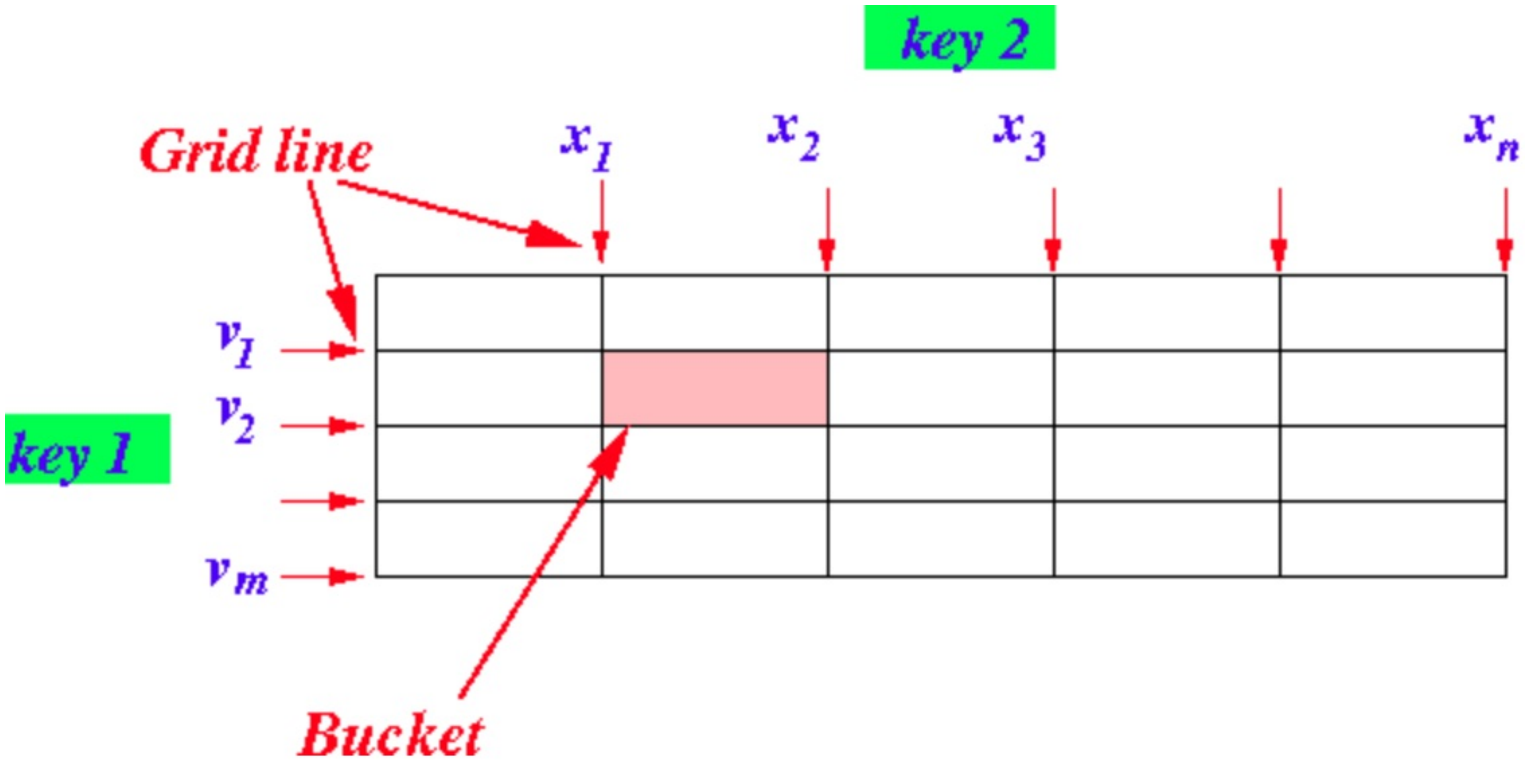
- Note: `Geographically` "`related`" `data` (i.e.: points that are close to each other) are stored in the same `data block`

Storage Structure of Grid Index File

1) A grid index file first stores the size parameters m and n of the grid
2) Then the index file stores the `buckets` of the grid
   - $v_1, v_2, \ldots, v_m$
   - $x_1, x_2, \ldots, x_n$
3) Finally, the index file contains $m \times n$ `block pointers`



Buckets and Grid lines

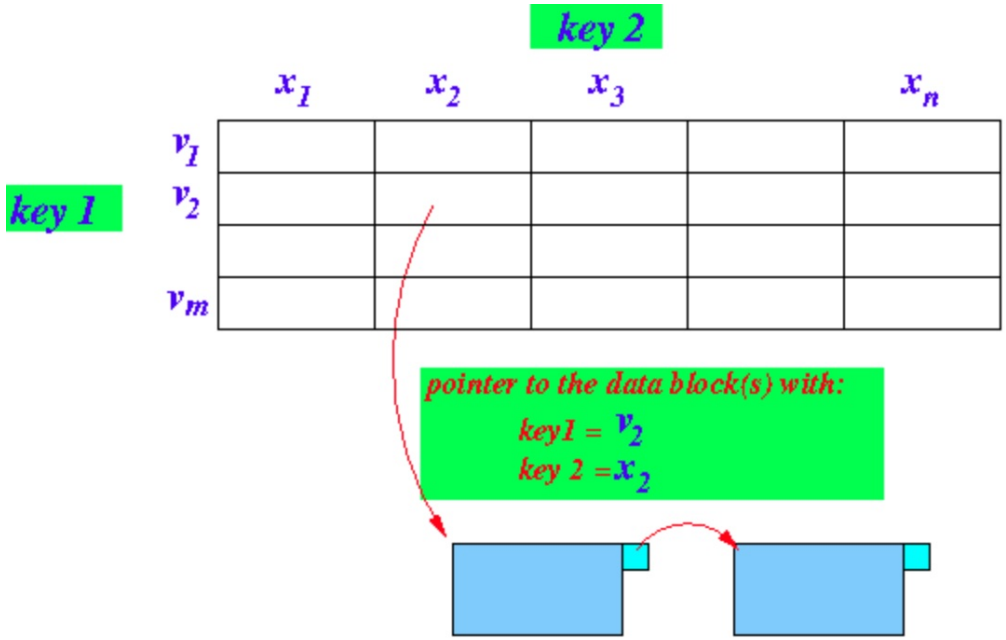


Interpreting the grid lines

- Two interpretations the grid lines
- You can interpret the values:
  - $v_1, v_2, \ldots, v_m$
  - $x_1, x_2, \ldots, x_n$
- in 2 ways:
  1) As individual points
  2) As intervals

Interpreting the grid lines: Point interpretation

- The bucket pointer in bucket (v,x) points to data blocks that store search key values:
  - `key1` = v
  - `key2` = x

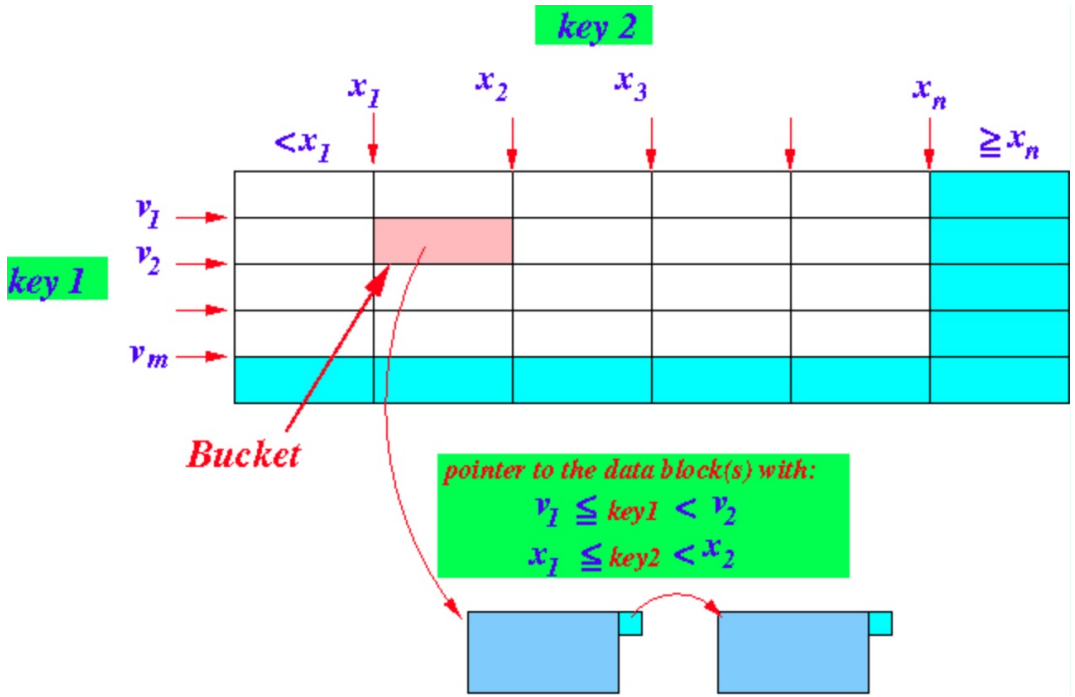


- The grid lines represents discrete values
- With n grid lines you will have n index points

Interpreting the grid lines: Interval interpretation

- The bucket pointer in bucket (v_i, x_i) points to data blocks that store search key values:
  - $v_{i-1} \leq$ `key1` $< v_i$
  - $x_{i-1} \leq$ `key2` $< x_i$

  (With additional edge buckets that covers the entire range of values)



- The grid lines represents end points of intervals
- With n grid lines you will have $n + 1$ intervals

Example of a Grid index file

- Data on `people` who buy jewelry:

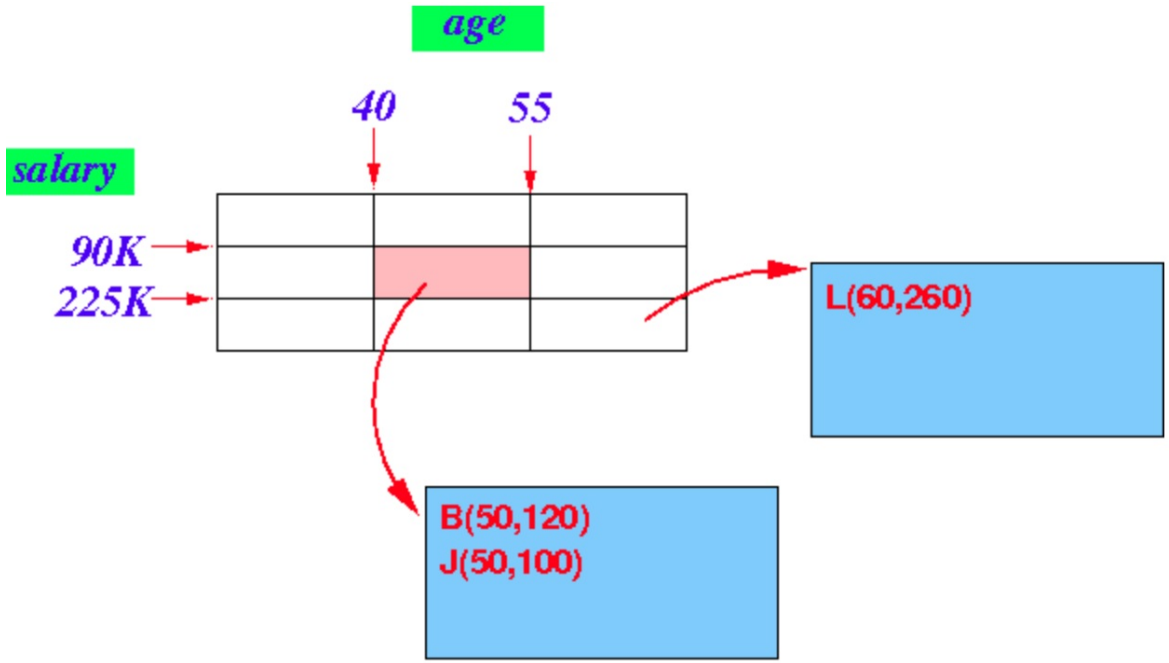
(age, salary (in $1,000))

A(25,60)	D(45,60)	G(50,75)	J(50,100)
B(50,120)	E(70,110)	H(85,140)	K(30,260)
C(25,400)	F(45,350)	I(50,275)	L(60,260)

- Ranges

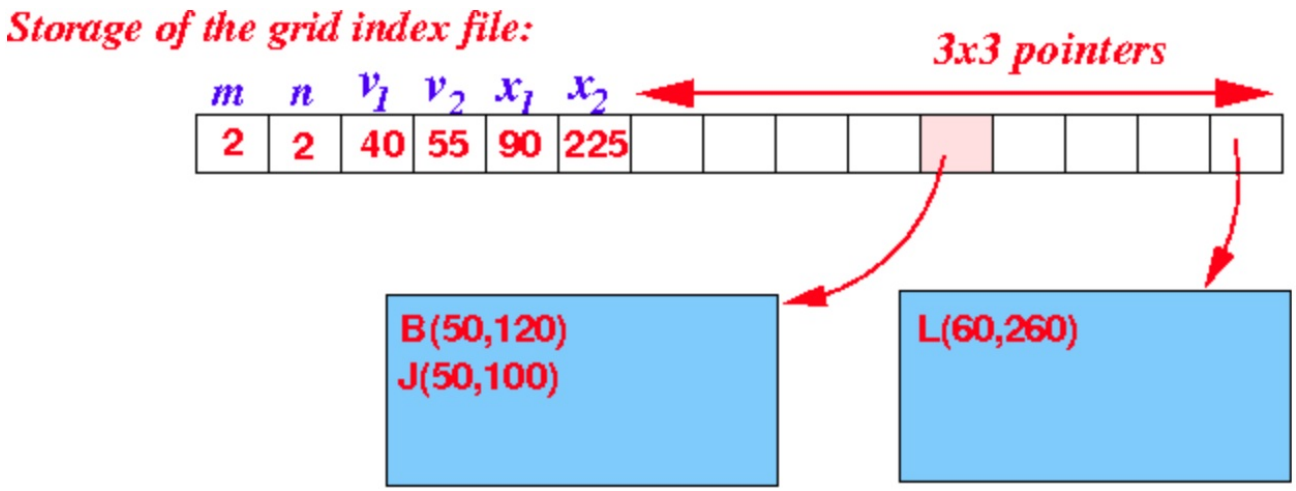
Age:	0-40	40-55	≥ 55+
Salary:	0-90K	90K-225K	≥ 225K+

- Grid index file



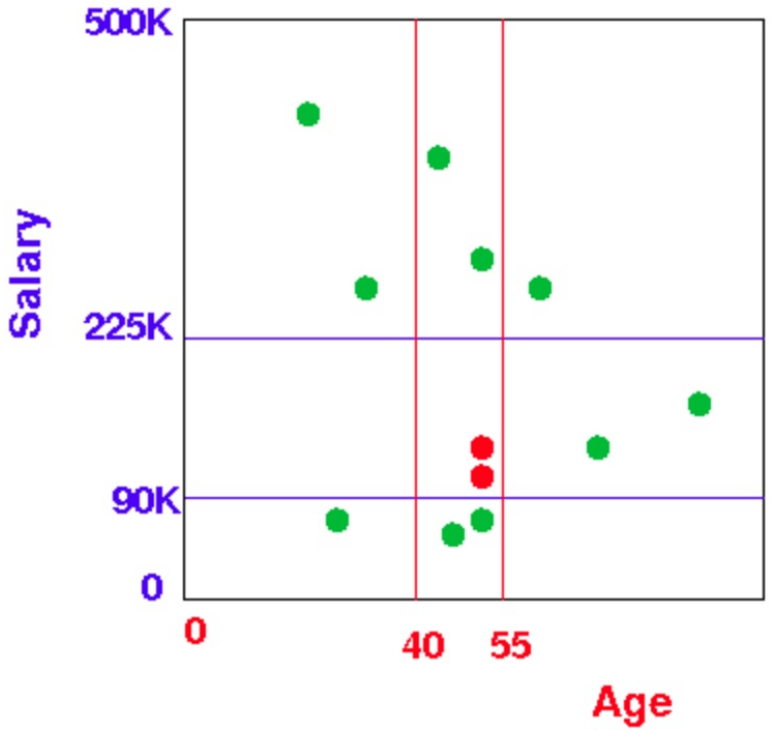
Example of a Grid index file

- How the grid index file is stored:



Example of a Grid index file

- The textbook use the following method to represent the index file



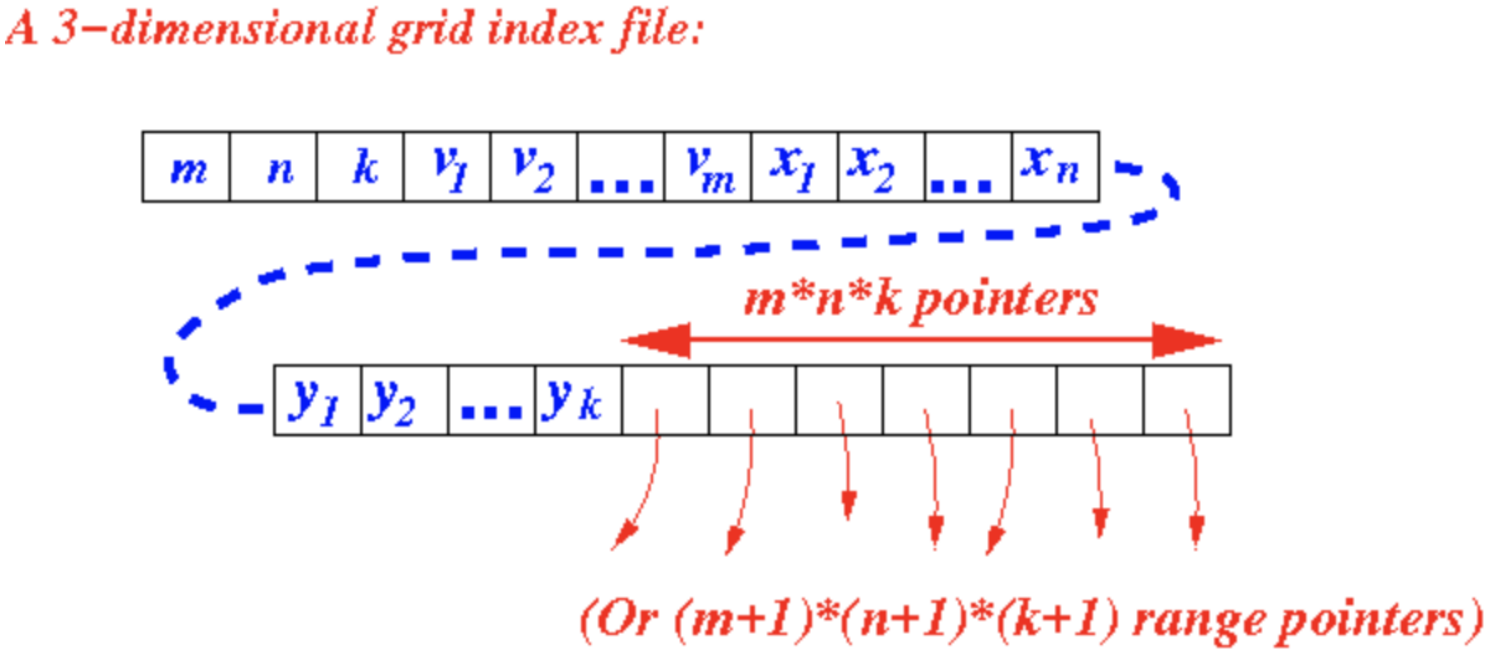
- For the following data set

(age, salary (in $1,000))

A(25,6)	D(45,60)	G(50,75)	J(50,100)
B(50,120)	E(70,110)	H(85,140)	K(30,260)
C(25,400)	F(45,350)	I(50,275)	L(60,260)

Generalization to higher dimensions

- It is easy to generalize the Grid Index to higher dimensions:



Lookup a search key

- **Given:** `search key (Age = 50, Salary = 100)`
- `How to find this record`
  - To locate the proper bucket for a point, we need to know, for each dimension, the list of values at which the grid lines occur.
  - Look at each component of the point and determine the position of the point in the grid for that dimension.
  - The positions of the point in each of the dimensions together determine the bucket.

Insert a new record in a Grid Index file

Algorithm 1 insert(record)

```
1: Lookup (record.SearchKey)
2: Let b = the last bucket block
3: if b has room for record then
4:     Insert record in block b
5: else
6:     Allocate an overflow block for bucket
7:     Link overflow block to b
8:     Insert record in overflow bucket block
9: end if
```

Performance Analysis: lookup/insert a search key

- Assumption: The grid index file can be store in memory
- **Lookup performance**
  - 0 `block access` to obtain the `bucket block` pointer
  - 1 `block access` to obtain the `data block` (that contains the record)
  - If there are `overflow blocks`, need to access a few more `(overflow) blocks`

Performance Analysis: lookup/insert a search key

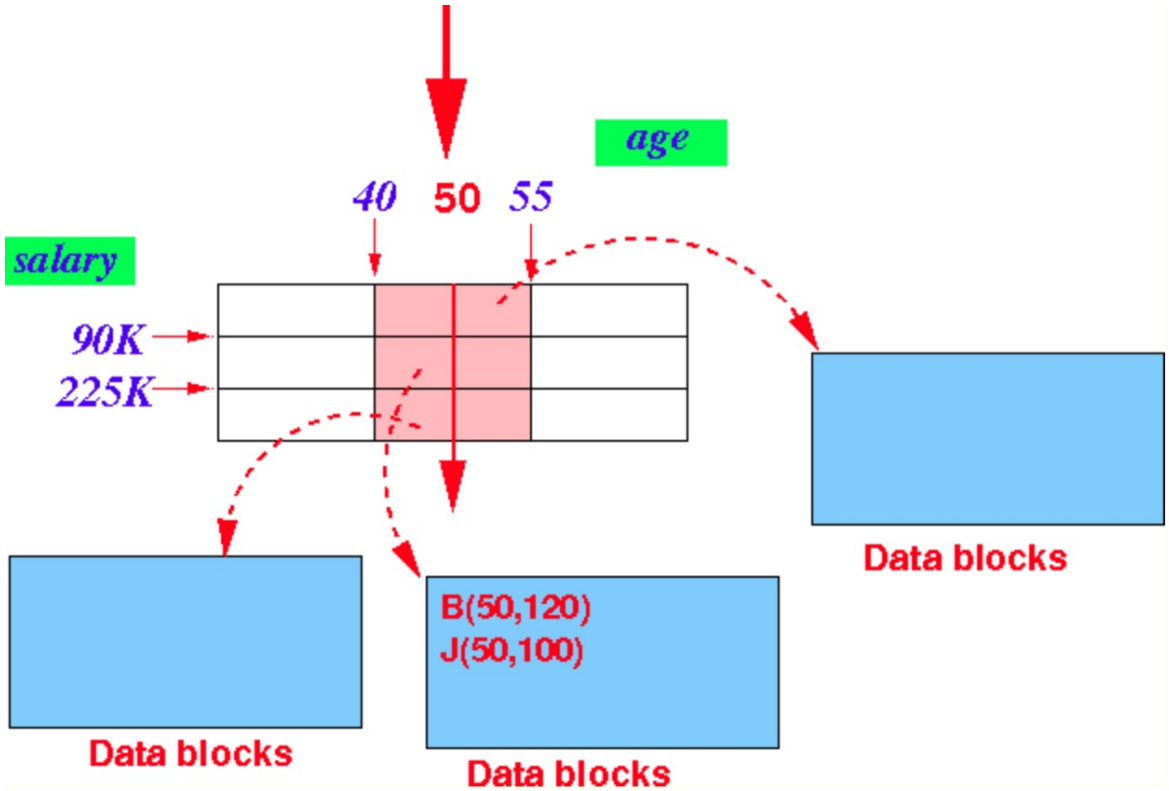
- **Insert performance**
  - In addition to the lookup cost
  - 1 more `block write operation` to update `the bucket block`
  - If `overflow`, need to update the overflow link in the `bucket` and write an `overflow block`

Using a grid index in multi-dimensional queries

- Performance of Grid index for the commonly used multi-dimensional queries
- Assumption: The grid index file can be stored entirely in memory
  - i.e.: there is no disk accesses needed to search the grid index file

1) Partial Match queries

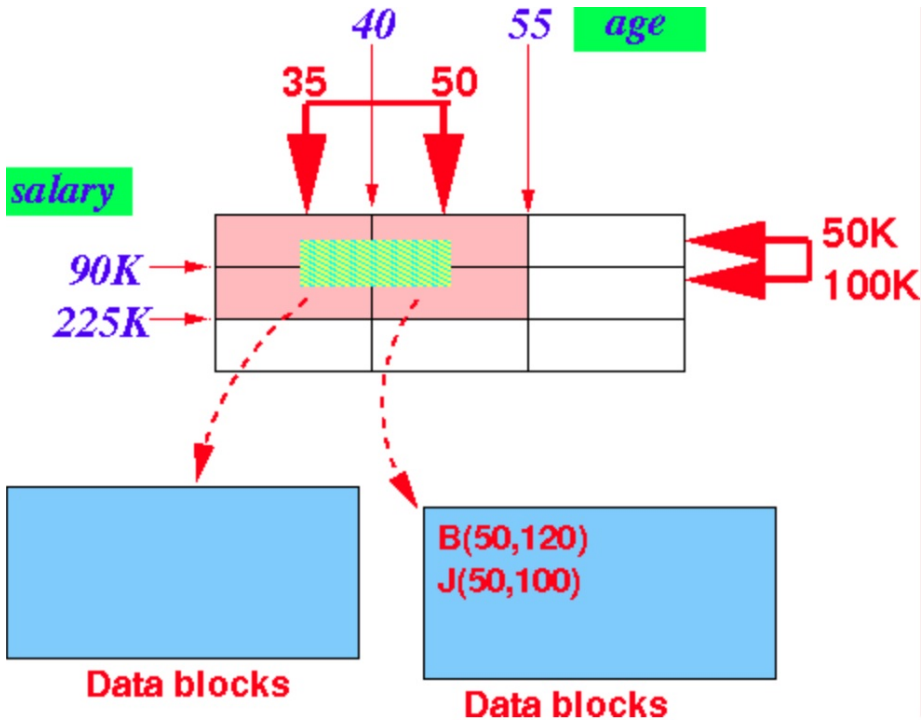
- `The query specifies conditions on some dimensions but not on all dimensions`
- Find all jewelry purchases by people with `age = 50`



- You will access m disk blocks (m is some dimension of the grid)
- **Note**: We access fewer disk blocks because "geographically related" data are stored in the same grid bucket

2) Range queries

- `Find objects that are located either partial or wholly within a certain range`
- Find all jewelry purchases by people whose $35 \leq age \leq 50$, $50K \leq salary \leq 100K$



- In this example, we must access 4 `disk blocks`

3) Nearest neighbor queries

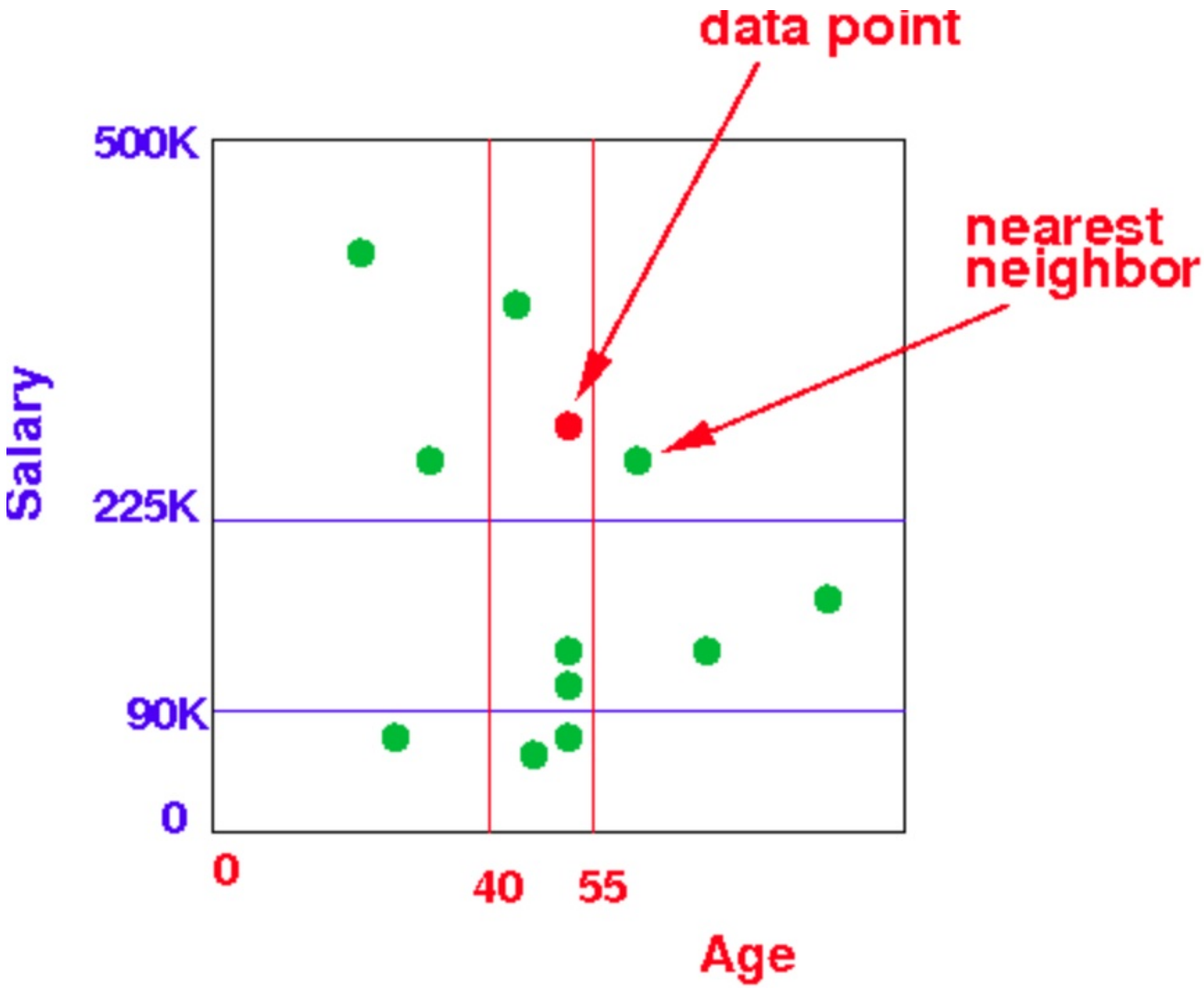
- Suppose we have a relation containing points on a map
- Each point is stored in the following relation: `Point(x,y)`
- Given a point p_0`(10,20)`
- Query: Find the point that is closest to point p_0`(10,20)`

```
SELECT * FROM   Point P
/* P is the closest point to p0 */
WHERE NOT EXISTS
            (SELECT *  FROM Point Q
           /* A closer point Q cannot exist ! */
            WHERE "distance(Q,p0)<distance(P,p0)")
```

```
SELECT * FROM Point P
/* P is the closest point to p0 */
WHERE NOT EXISTS
            (SELECT *  FROM Point Q
           /* A closer point Q cannot exist ! */
            WHERE (Q.x-10)^2+(Q.y-20)^2<(P.x-10)^2+(P.y-20)^2)
```

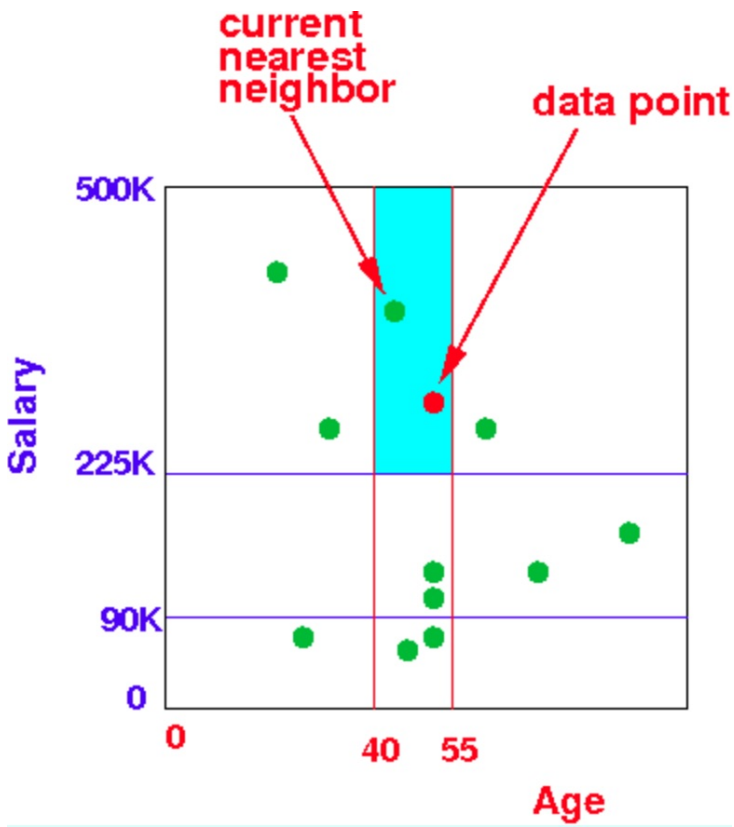
3) Nearest neighbor queries

- Find the nearest neighbor of a data point



3) Nearest neighbor queries

- Start by finding the `nearest neighbor` in the `bucket` that contains the `data point`

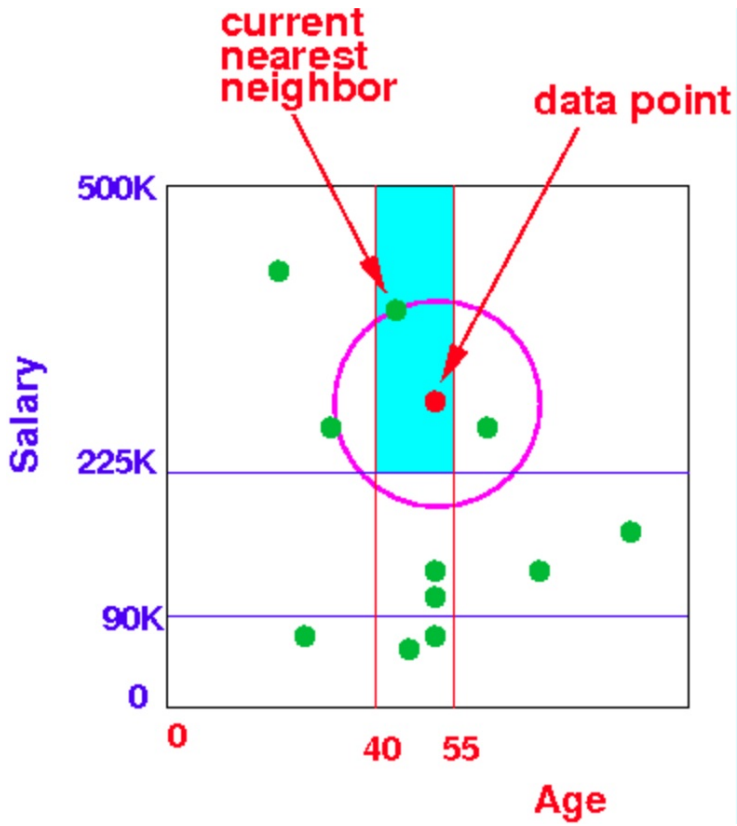


- Nearest neighbor candidate:
  - Nearest neighbor candidate: the nearest neighbor found so far in the search
  - Search distance: the distance of the nearest neighbor candidate to the search data point

3) Nearest neighbor queries

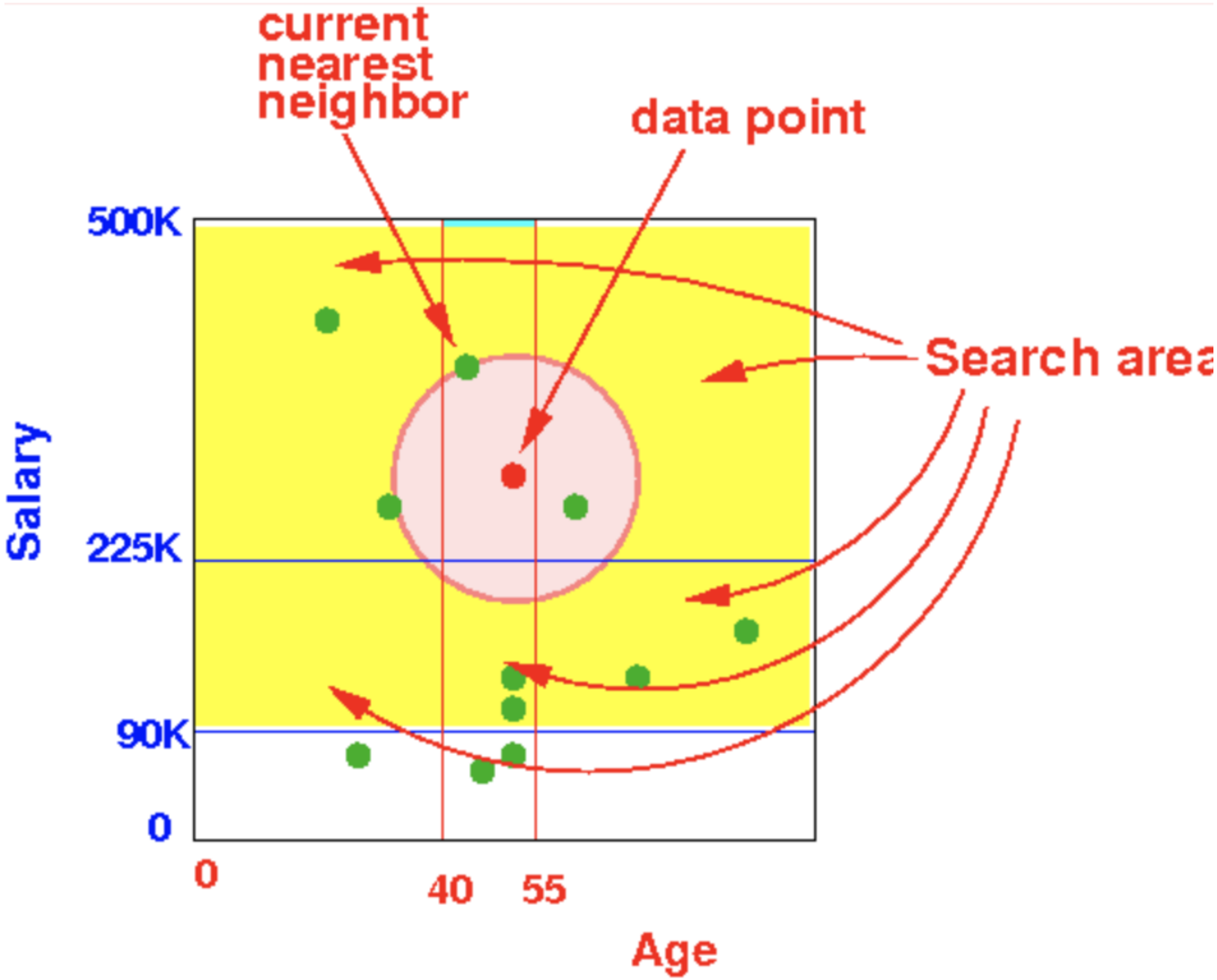
- **Limit the search space**
  - Use the search distance to limit the grid index buckets where you need to search to find the nearest neighbor
    - Draw a circle around the data point

$$(age\text{-}age_p)^2 + (salary\text{-}salary_p)^2 \leq search-distance^2$$



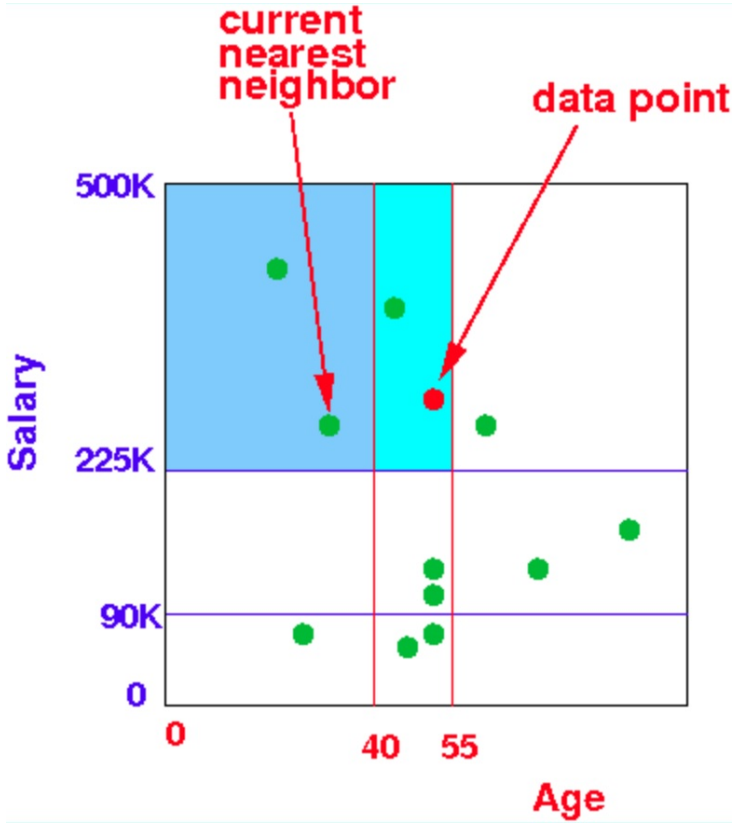
3) Nearest neighbor queries

- Limit the search space to the grid index buckets that intersects with the circle



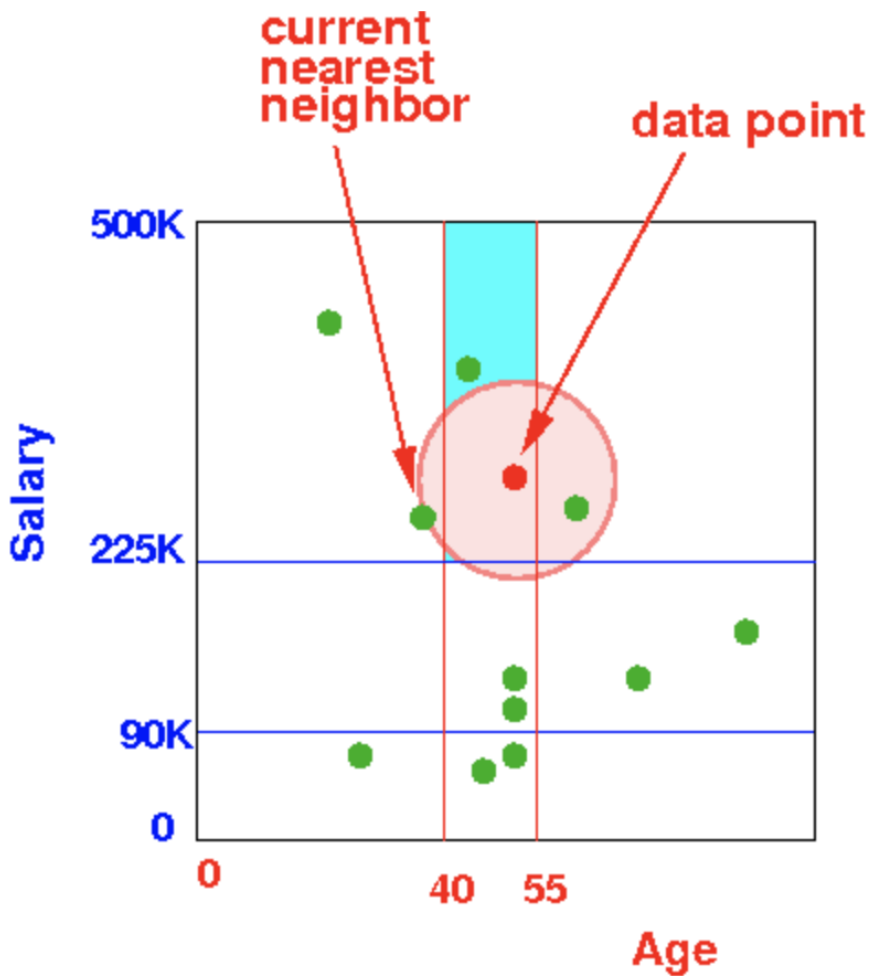
3) Nearest neighbor queries

- Proceed to search in the first "candidate" grid index bucket that is intersects the circle:
  - When you find a better nearest neighbor



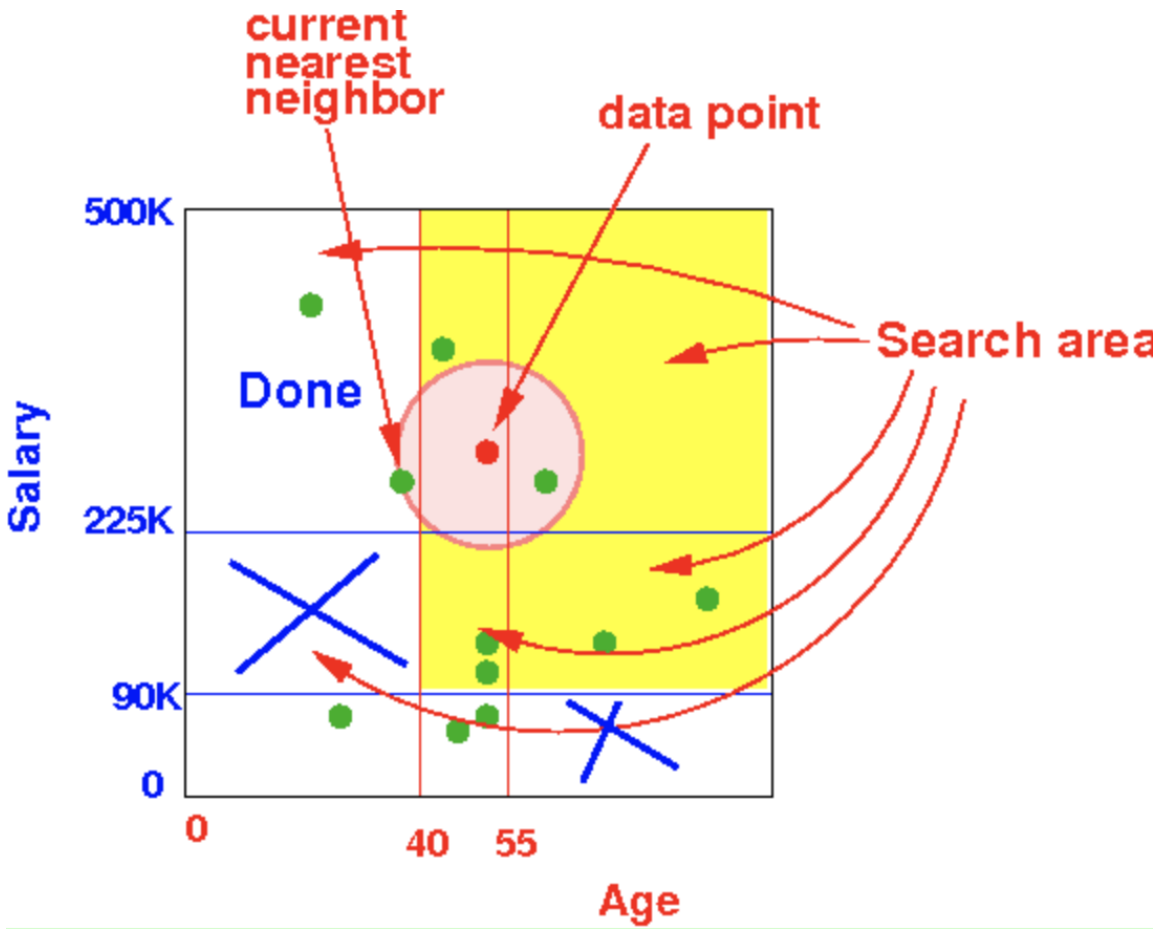
3) Nearest neighbor queries

- Proceed to search in the first "candidate" grid index bucket that is intersects the circle:
  - When you find a better nearest neighbor, re-compute the (smaller) search distance



3) Nearest neighbor queries

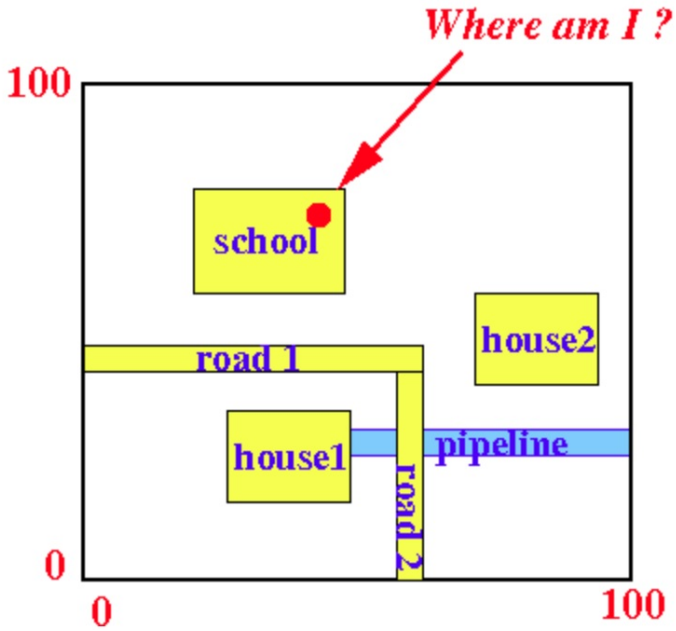
- Proceed to search in the first "candidate" grid index bucket that is intersects the circle:
  - When you find a better nearest neighbor, re-compute the (smaller) search distance, and limit the search space further:



- And so forth

4) Where-am-I queries

- Given a `location (i.e., coordinate)`
- Find the `object(s)` that contains the `location`



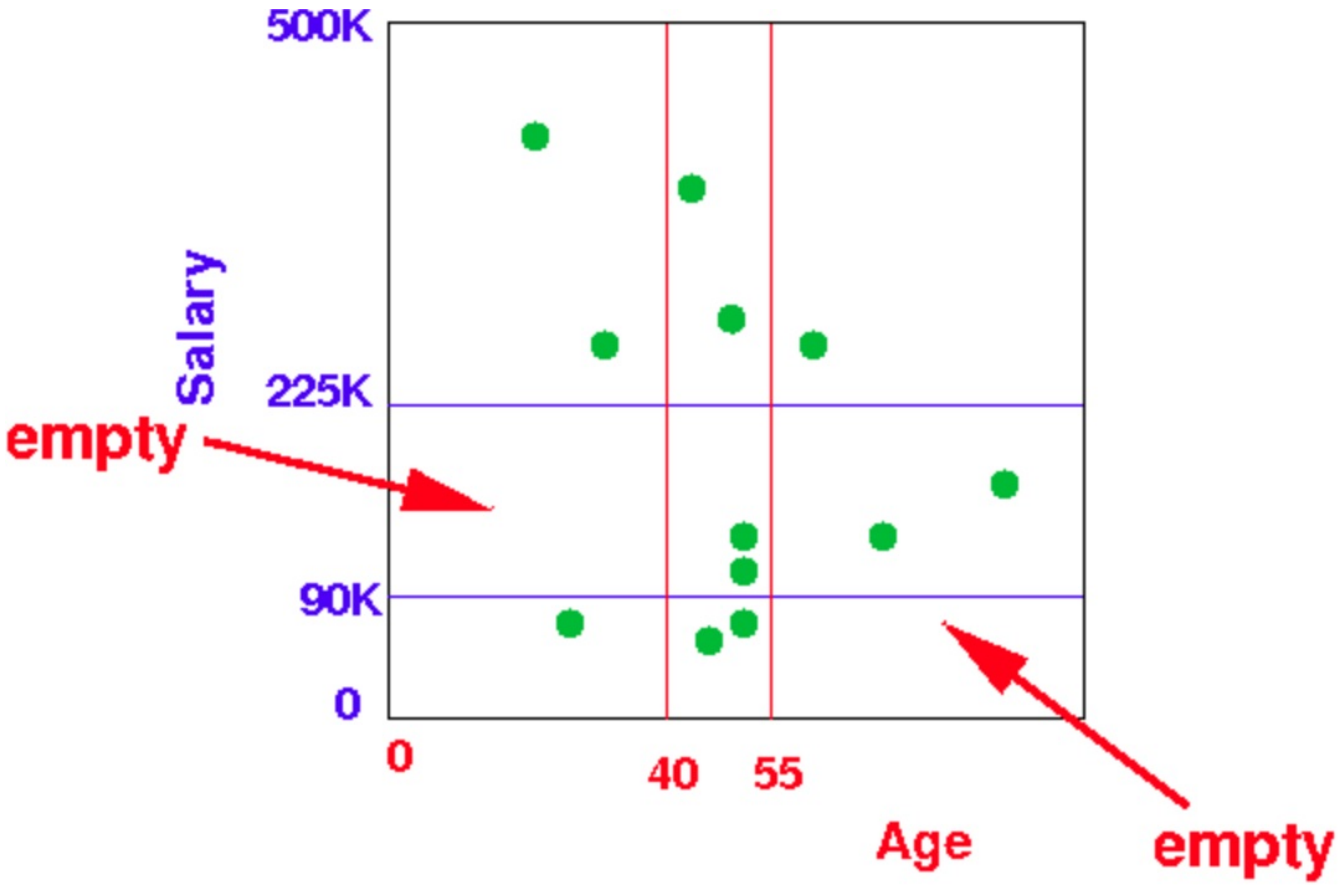
- `Grid index` cannot represent objects (can only represent points)
- ⇒ `Grid Index` cannot handle `Where-am-I` type of queries
- The only kind of index that can handle `Where-am-I queries` is `the R-tree (Region-tree)` (Discussed later)

Grid Index: Summary

- What can we do if a block does not have more room?
  - allow overflow blocks
  - add more grids
- Grid files can quickly find records
  - $key1 = V_i \ AND \ Key2 = X_j$
  - $key1 = V_i$
  - $key2 = X_j$
  - $key1 \geq V_i \ AND \ key2 < X_j$
- Grid files are
  - ☺ Good for multiple-key search
  - ☹ Space, management overhead (nothing is free)
  - ☹ Need partitioning ranges that evenly split keys

Grid Index

- A major problem with Grid Index files is `Poor occupancy rate at many grid buckets`



- Especially when you have 3 or more dimensions. You will have many buckets that are empty.

Multi-dimensional index structures

- Hash like structures
  - Grid files
  - Partitioned Hash functions
- Tree like structures
  - Multiple key indexes
  - kd-trees
  - Quad trees
  - R-trees

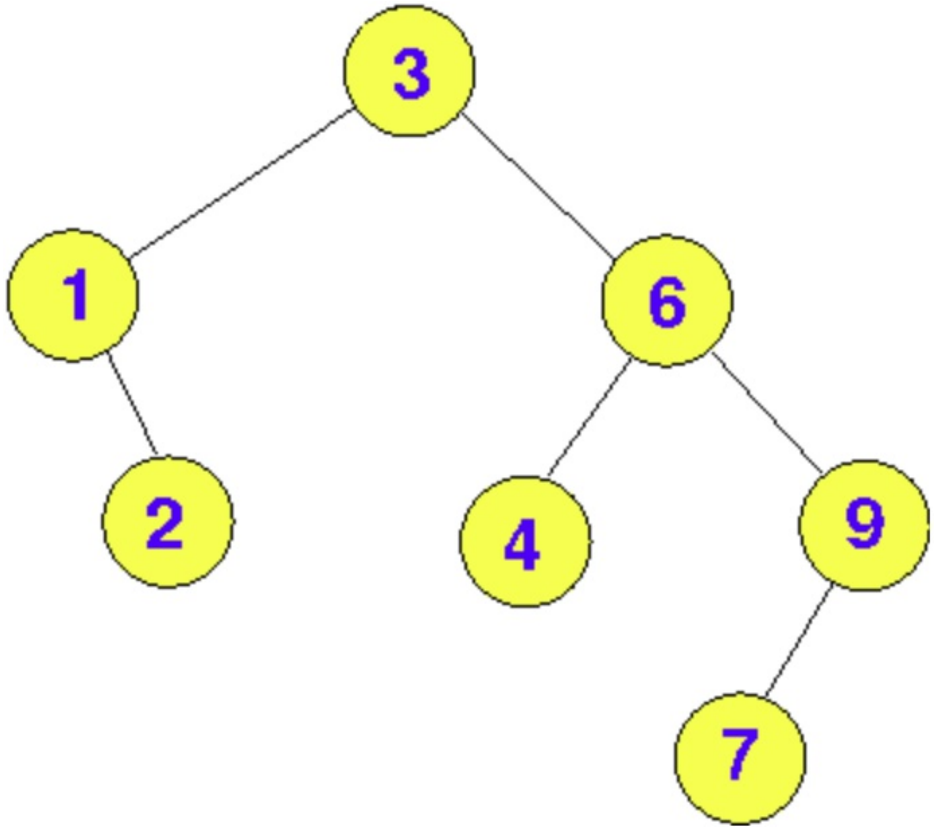
kd (k-dimensional) tree:

- The `kd-tree` as a main memory data structure
- Adaptation of the `kd-tree` for disk storage

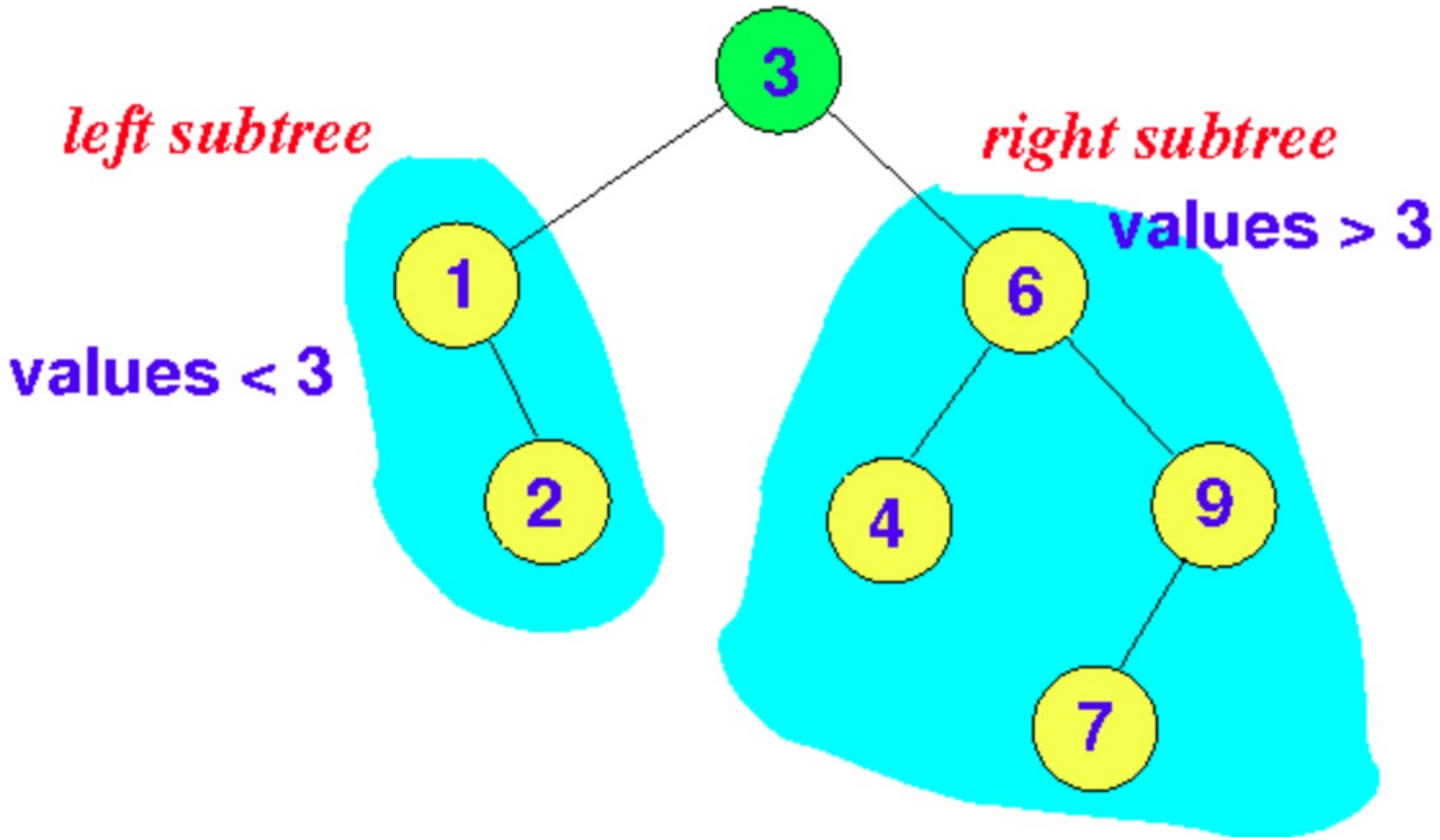
Review: Binary Search Tree

- `Binary Search Tree (BST)` is a `binary tree` where
  - The values in the nodes in the `left subtree` of the `node` x in the tree has a smaller value than x
  - The values in the nodes in the `right subtree` of the `node` x in the tree has a greater value than x
- Notice the above property holds for every node in the `binary tree`

Review: Binary Search Tree: Example



Review: Binary Search Tree: Example



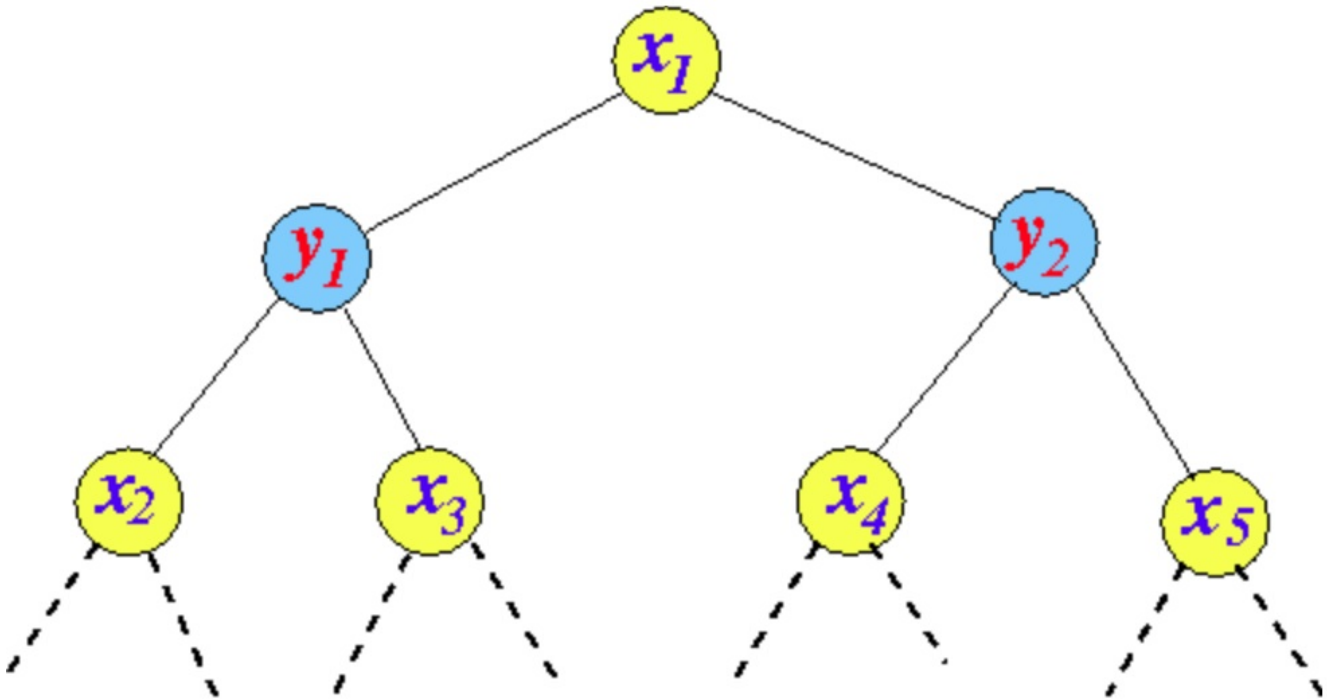
The kd-tree

- The `kd-tree` is a generalization of the classic `Binary Search Tree (BST)`
- The `search key` used at different levels belongs to a different dimension (domain)
- The dimensions at different levels will wrap around (i.e., circulate)

- Reading: Space-partitioned GiST, PostgreSQL

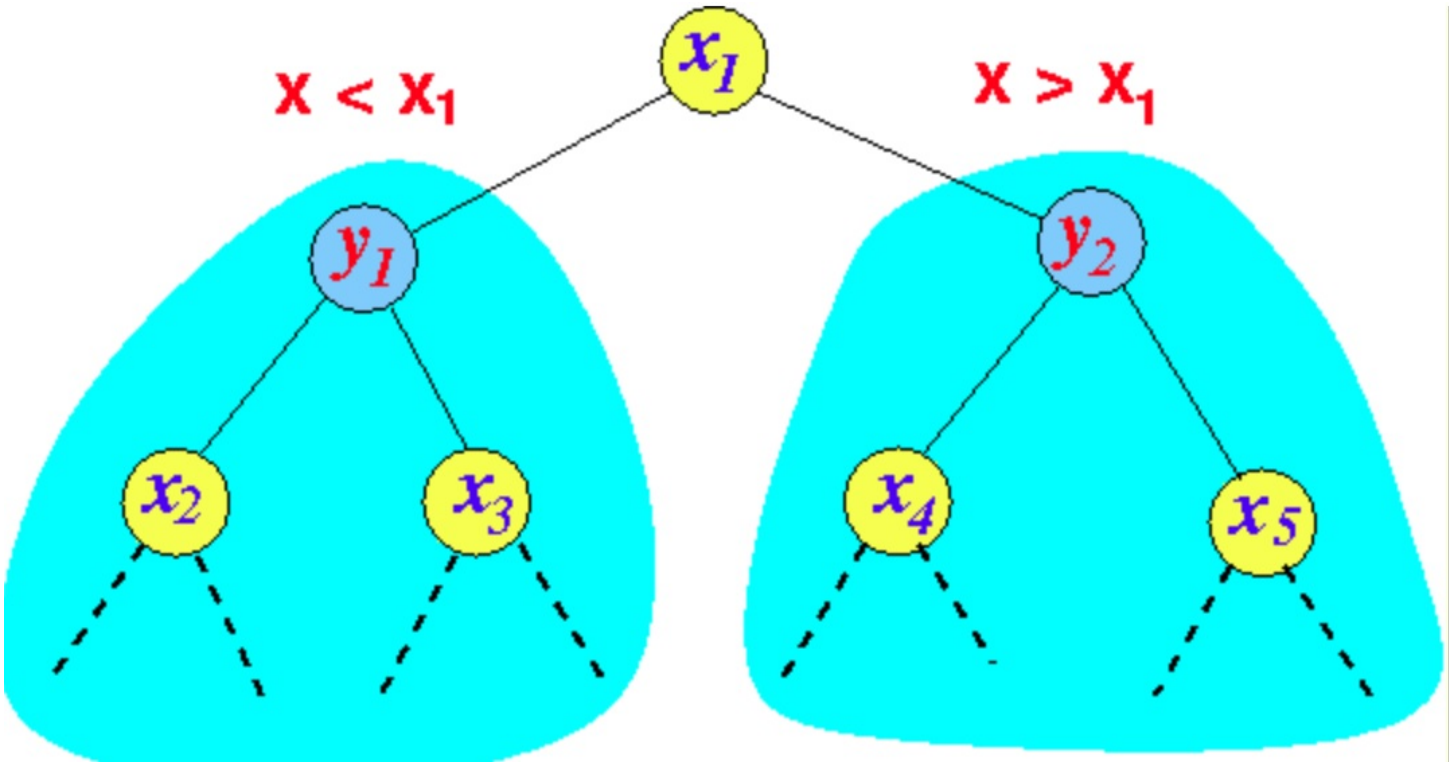
Example: a 2-dimensional kd-tree

- 2 dimensions: x and y



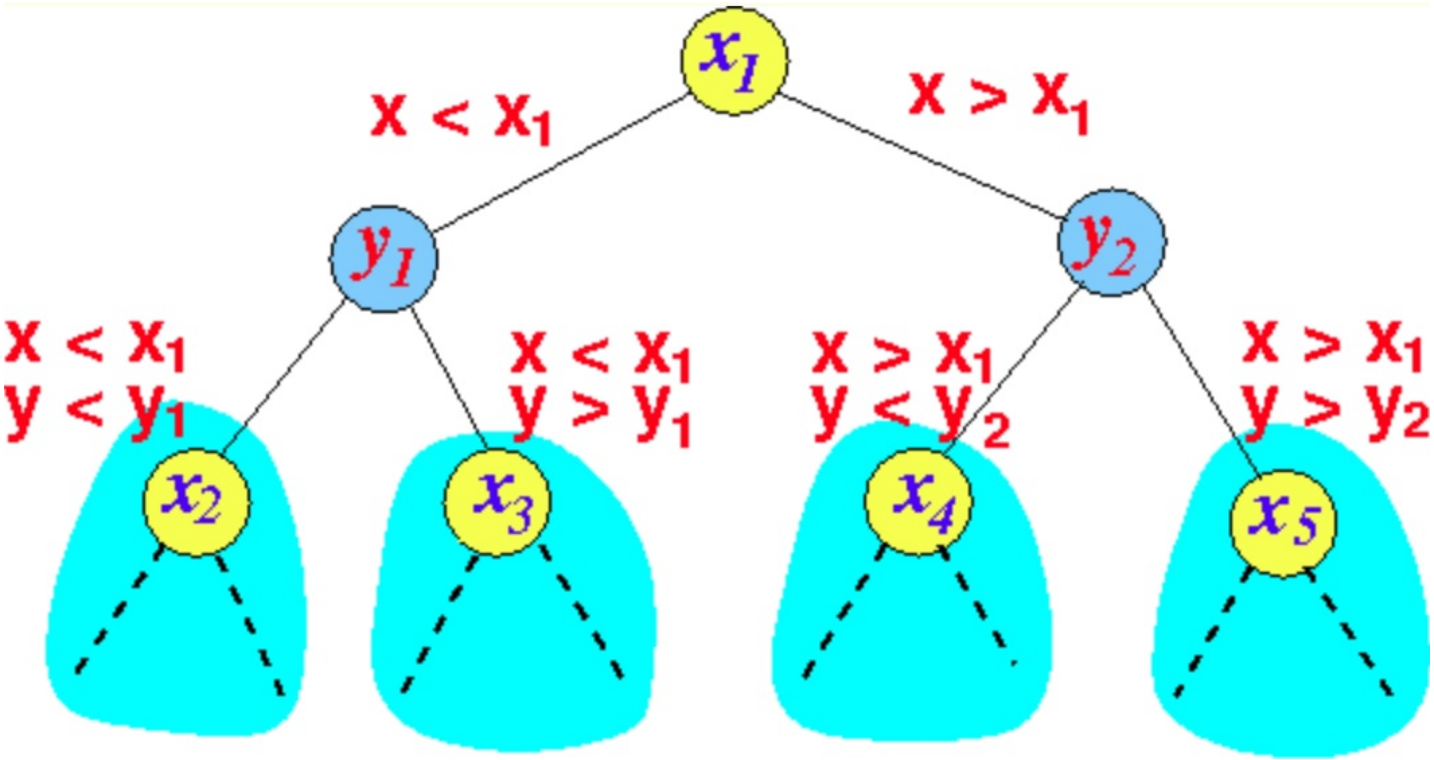
Properties

- `Subtrees` of x_1 must satisfy this property



Properties

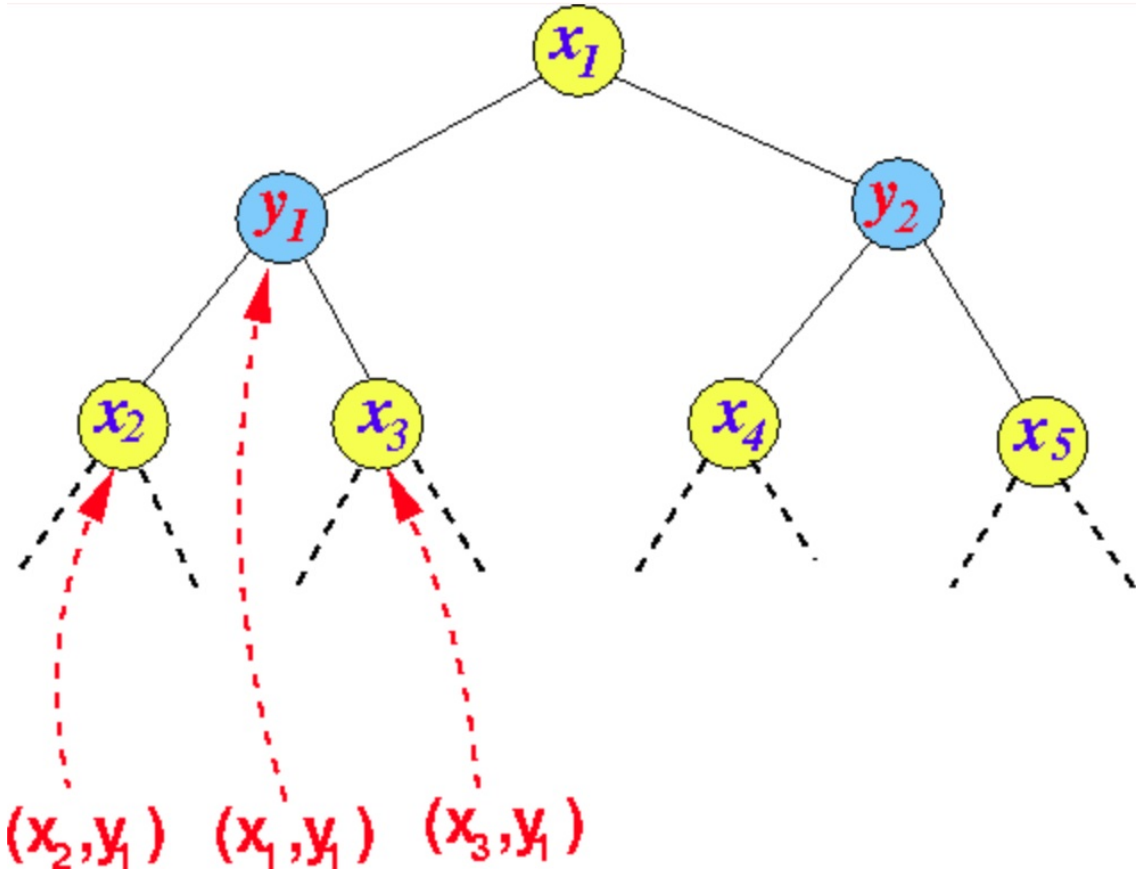
- `Subtrees` of y_1 and y_2 must satisfy this property



- And so on (for every level of the `kd-tree`)

Classical kd-tree

- The actual `record (data)` are stored in every `node (search key)` of the `kd-tree`



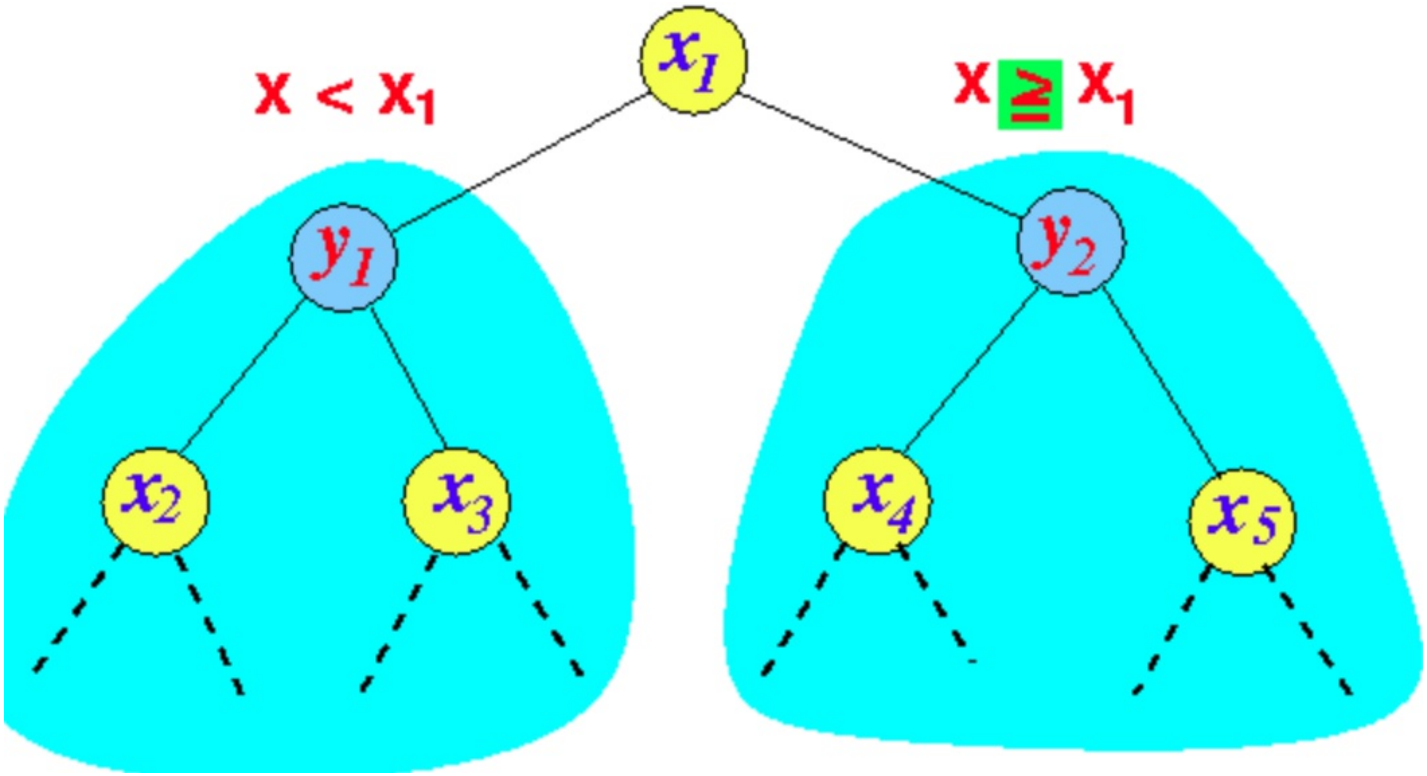
- The `node` y_1 contains the `data (record)` for (x_1, y_1)
- The `node` x_2 contains the `data (record)` for (x_2, y_1)
- And so on

Modifications to the `kd-tree` for storage on disk

- `Interior nodes` do not store data
- `Interior node` only stores
  - Attribute name (i.e.: `X` or `Y`)
  - Dividing value (i.e.: x_1 or y_4) of the attribute
  - Pointers to the (2) children nodes

Modifications to the `kd-tree` for storage on disk

- Dividing line is "moved" a little bit



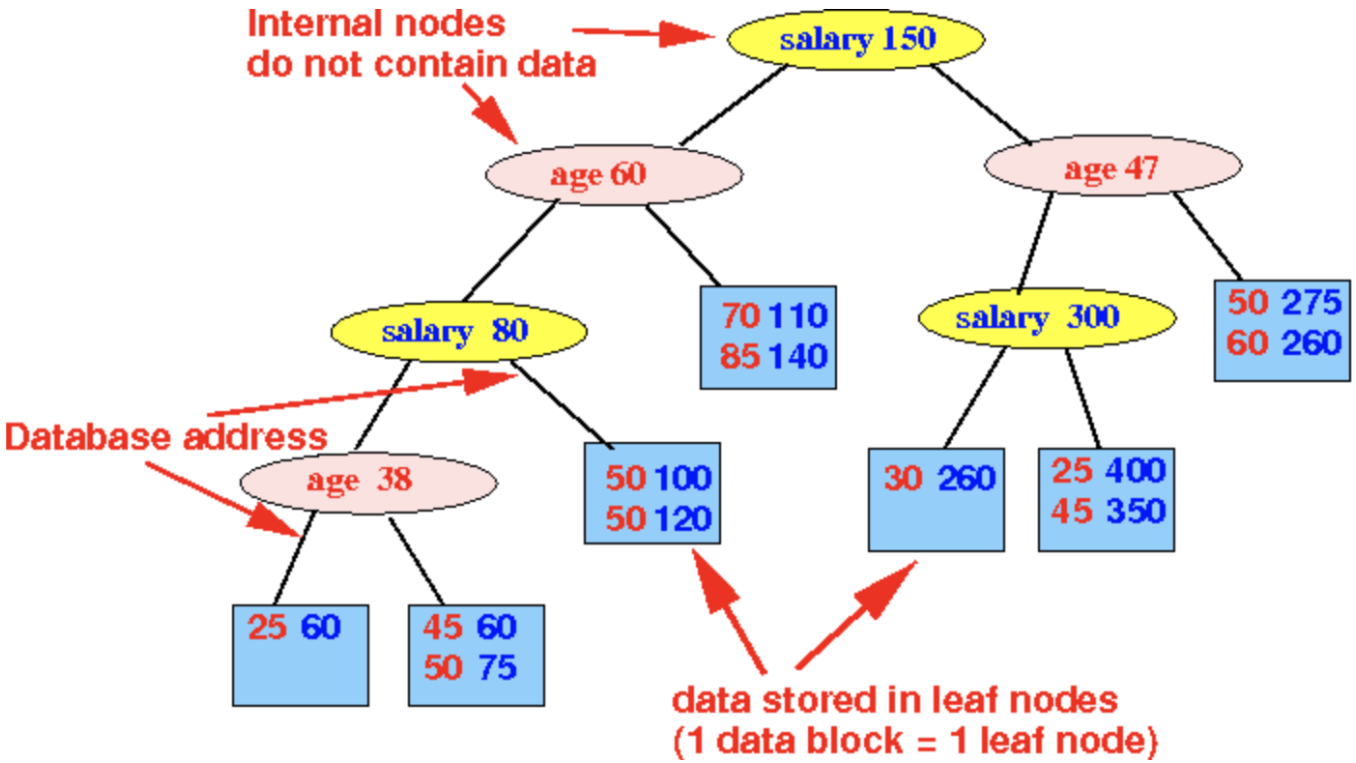
- The `equality` is included in right branch of the `kd-tree`
- Each `leaf node` of the `modified kd-tree` (for disk storage) is stored in one (1) `data block`

Example kd-tree

- Data on people who buy jewelry

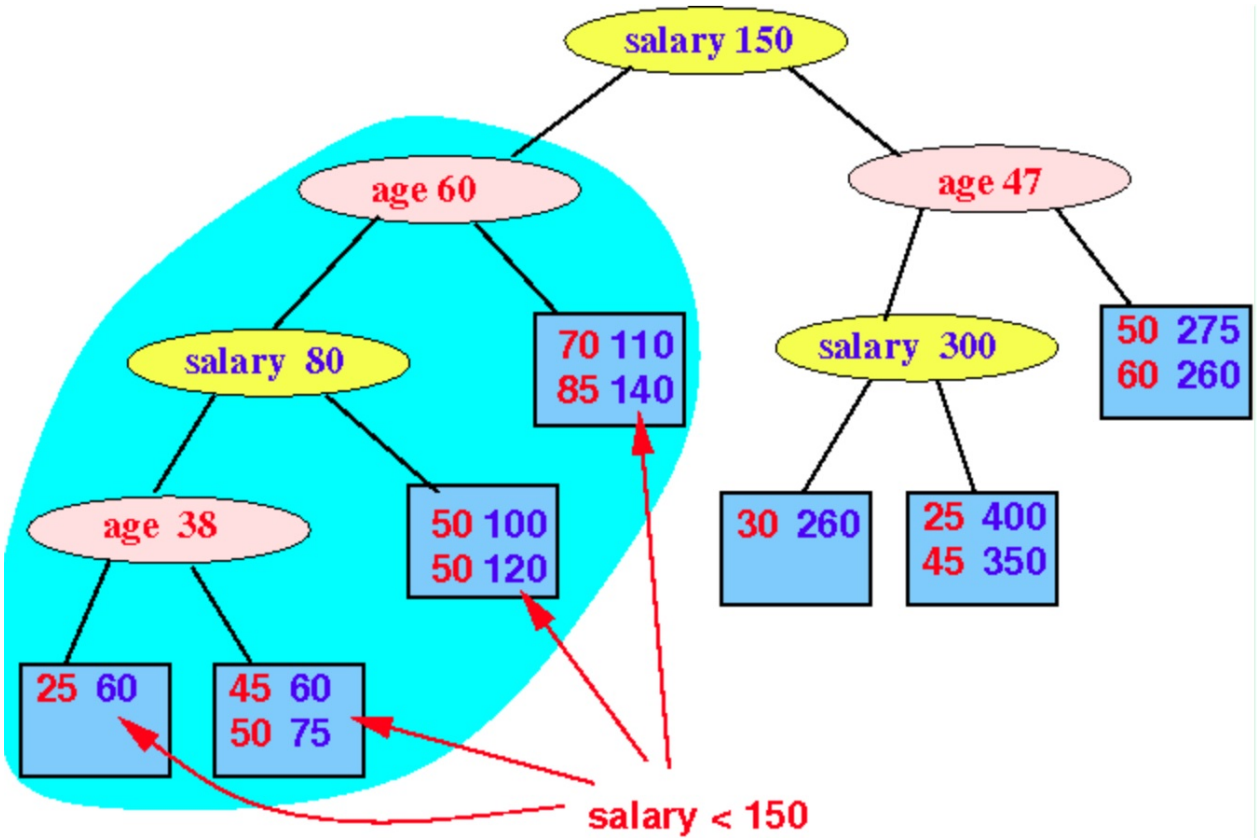
```
(age, salary (in $1,000))

A(25,60)     D(45,60)     G(50,75)     J(50,100)
B(50,120)    E(70,110)    H(85,140)    K(30,260)
C(25,400)    F(45,350)    I(50,275)    L(60,260)
```

- A kd-tree for the data:



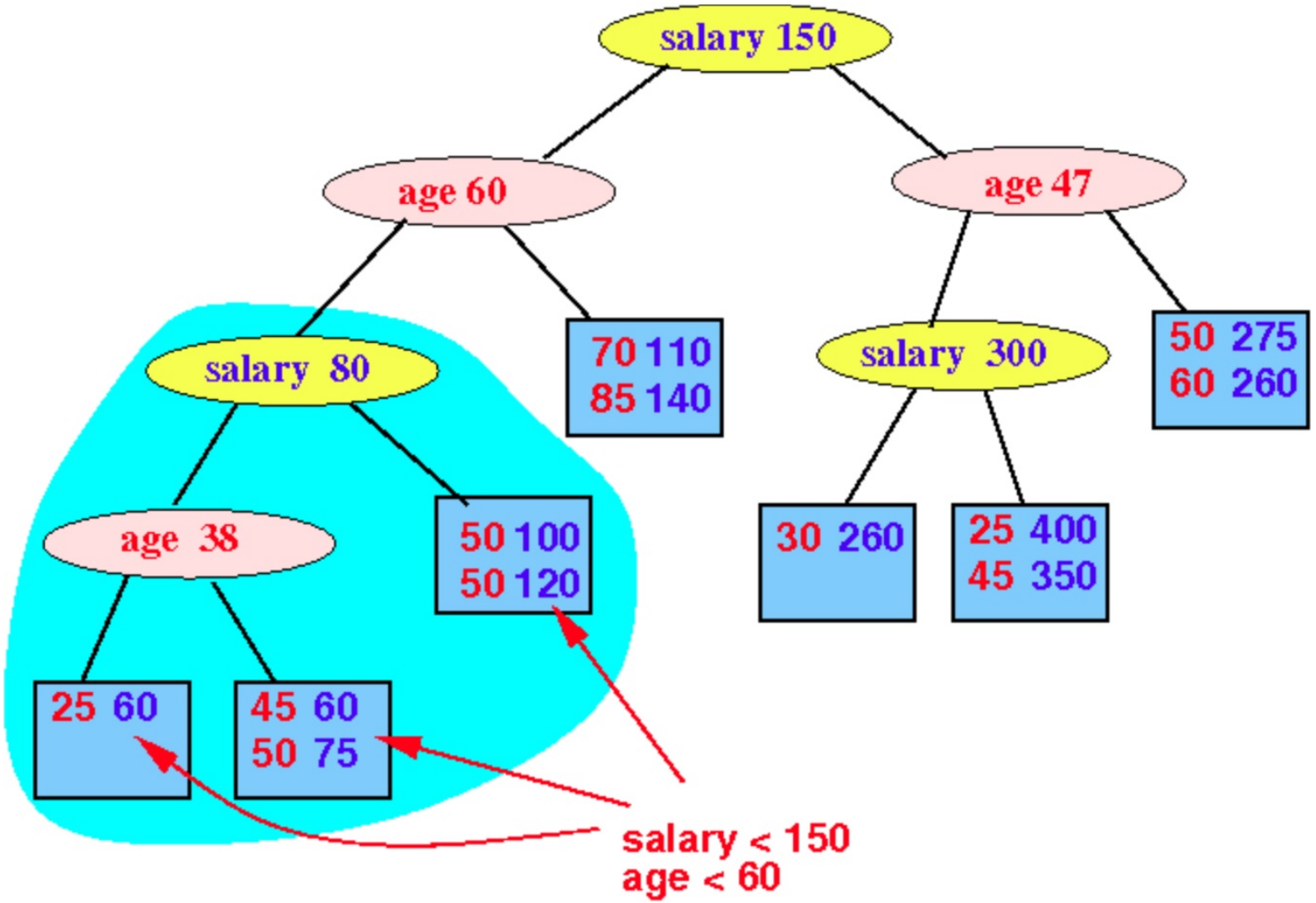
Search Property of a `kd-tree`

- The structural properties of the `kd-tree` that allows us to search in the `kd-tree`:
  - All data records in the `left (shaded) subtree` in the figure below has `salary search key` values < 150 (for salary):



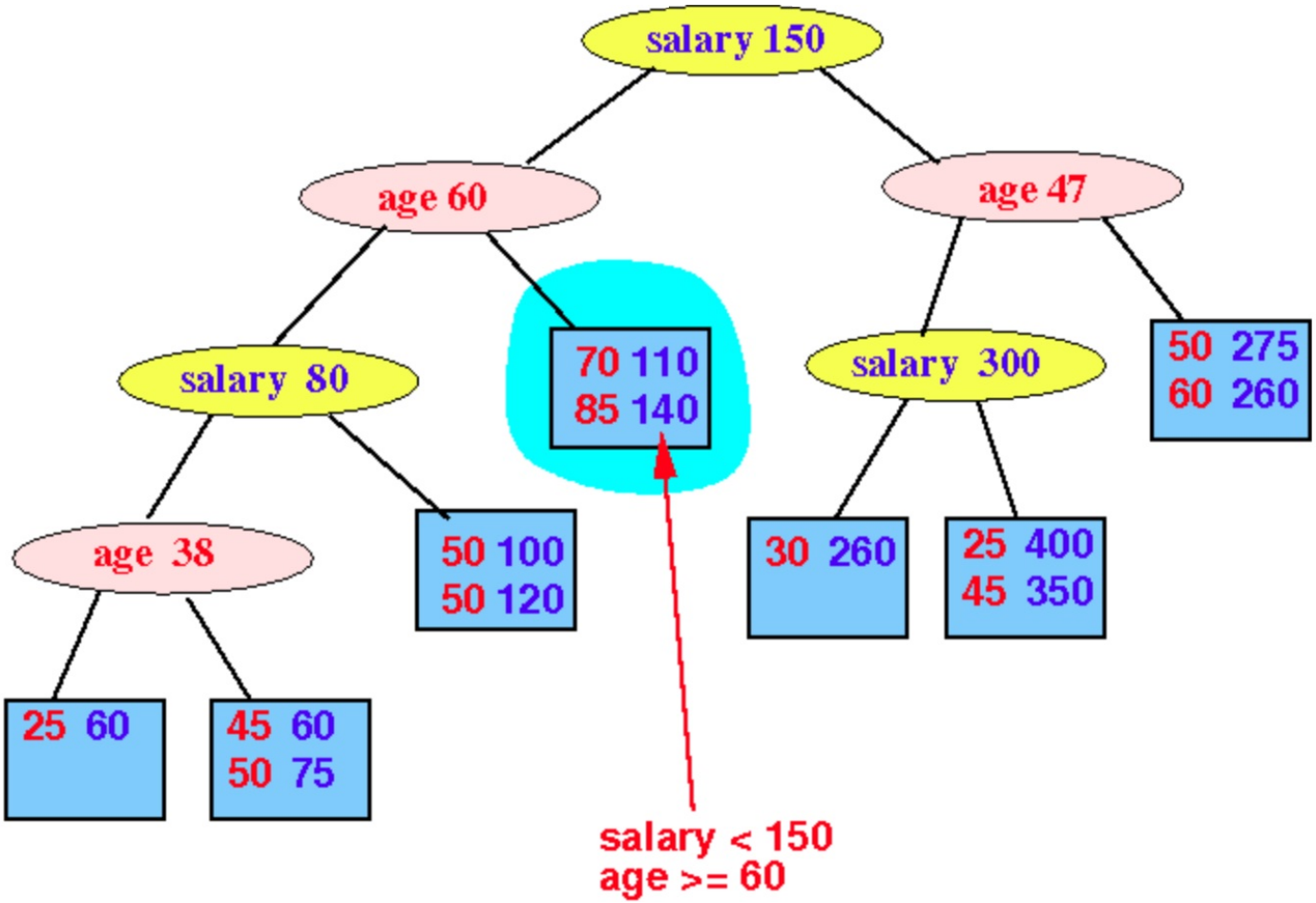
Search Property of a `kd-tree`

- Data records in this left subtree has has `salary` $<$ `150` and `age` $<$ `60`



Search Property of a kd-tree

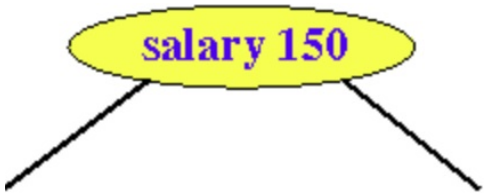
- This `right subtree` has `salary` $<$ `150` and `age` $\geq$ `60`



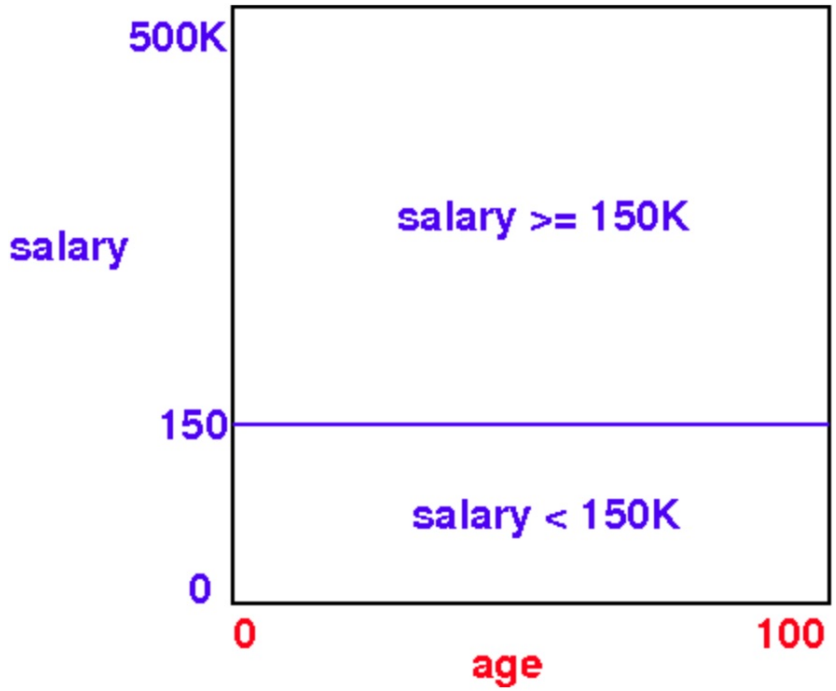
- And so on
- The search property is similar to the search property of the Binary Search Tree

How a `kd-tree` partitions the data space

- The internal nodes of the `kd-tree` partitions the search space into disjoint search areas
- The following figures shows how a `kd-tree` partitions the search space:
  - The `root node`

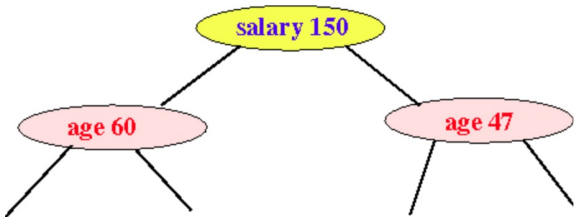


  - partitions the data space in half

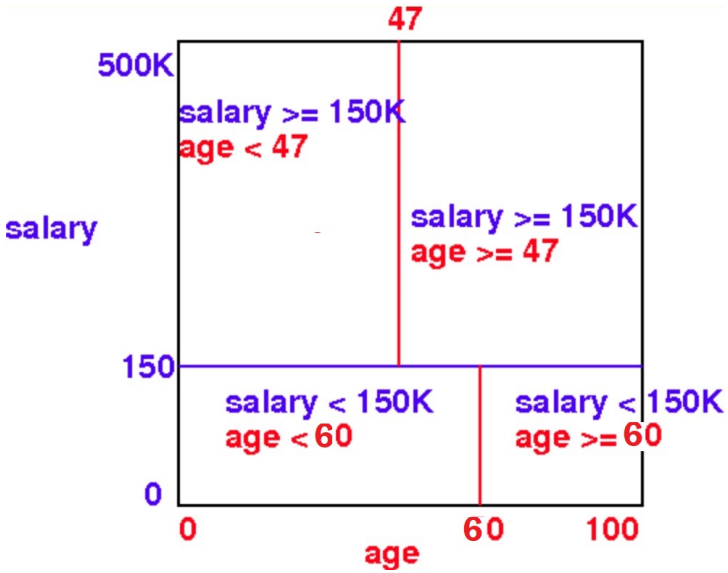


How a `kd-tree` partitions the data space

- The `age nodes` at `level 2`



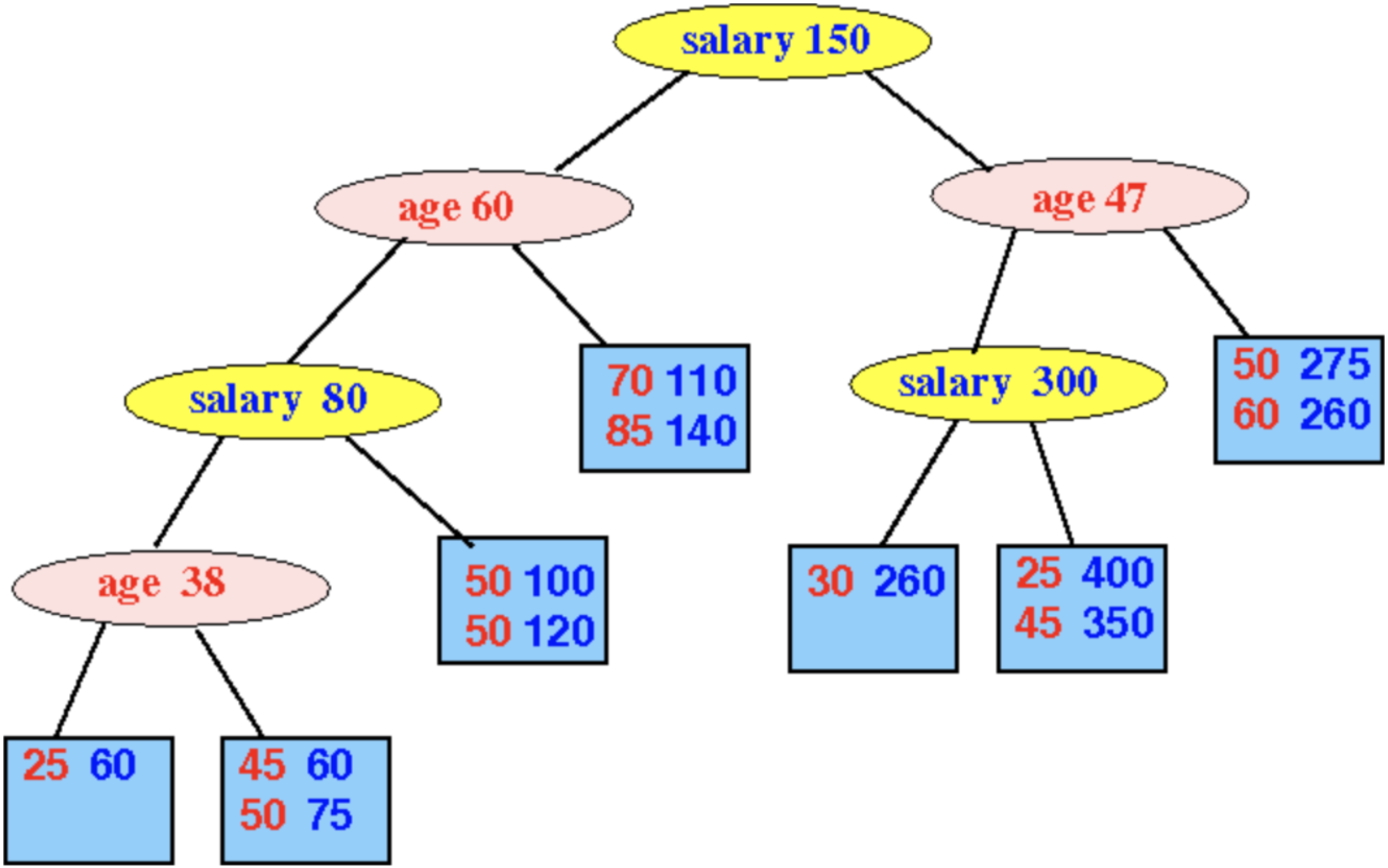
- partitions each sub-space in half



- And, so on...

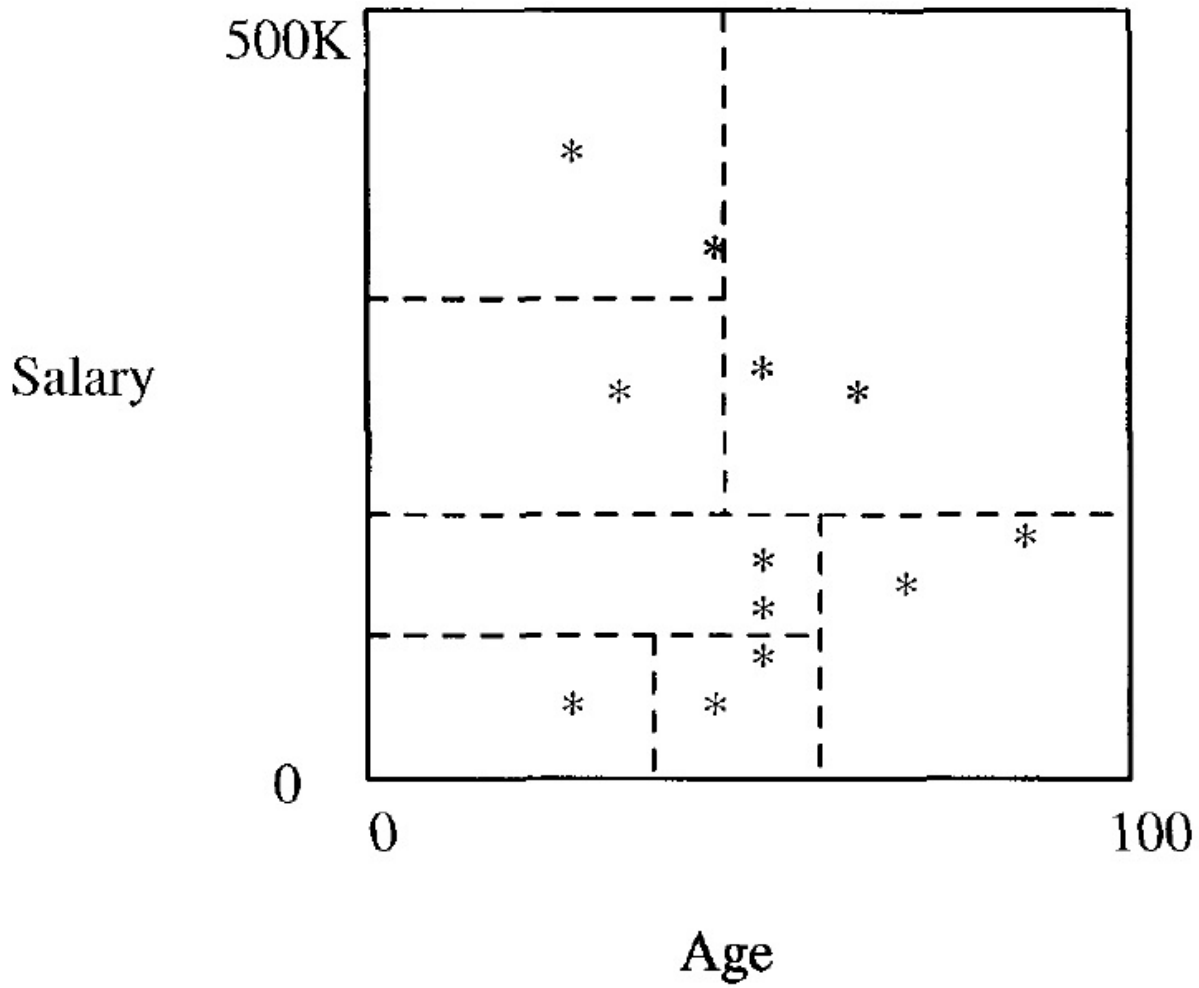
How a kd-tree partitions the data space

- This kd-tree



How a `kd-tree` partitions the data space

- will divide the data space up as follows



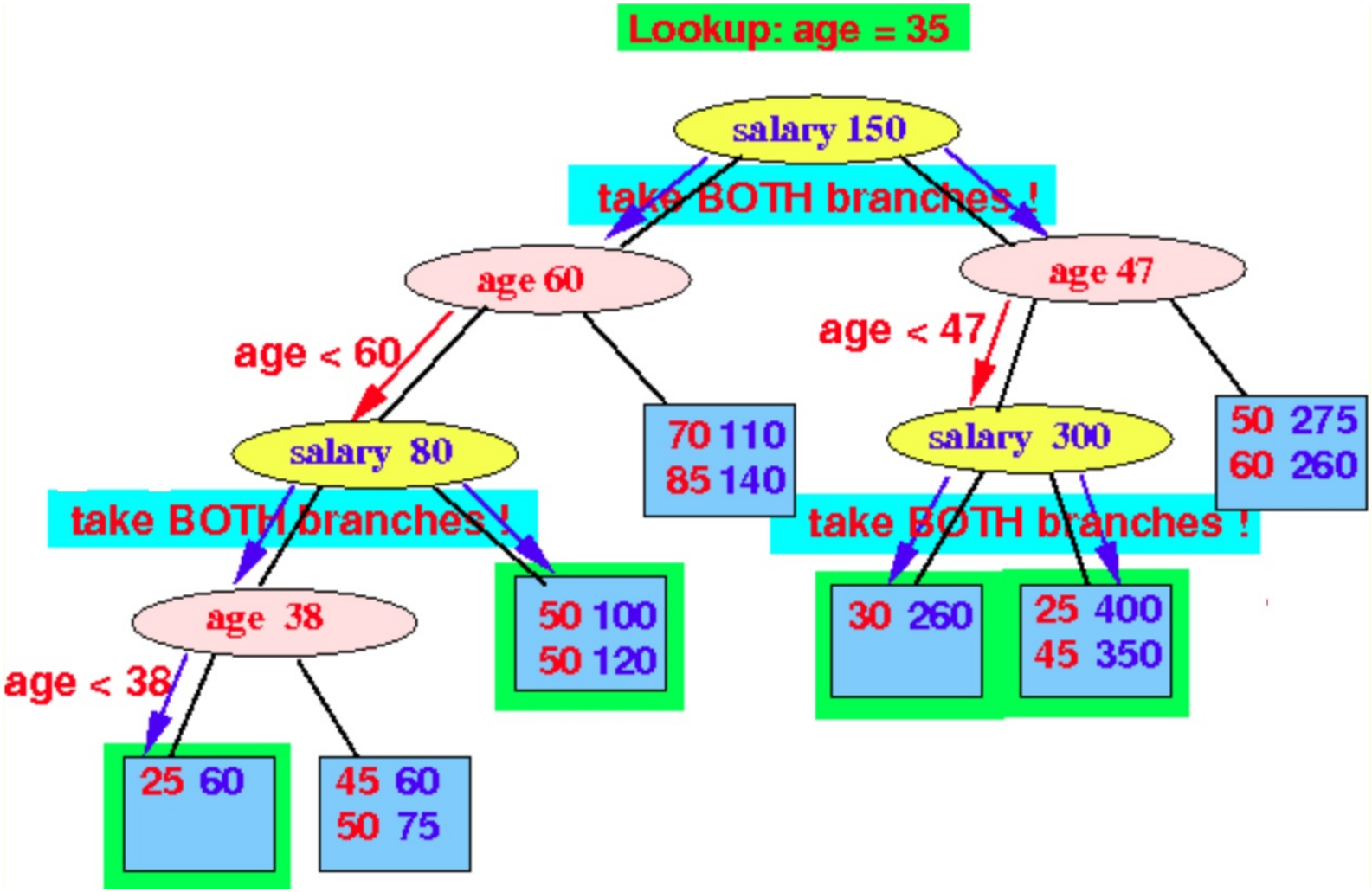
Using `kd-tree` for common multi-dim queries

1) `Partial Match queries`

- Search Algorithm
  - For a dimension for which the search value is given (specified)
    - Take the (one) branch of the subtree for the search value
  - For a dimension for which the search value is not given (not specified)
    - Take both branches of the subtree

1) Partial Match queries: Example

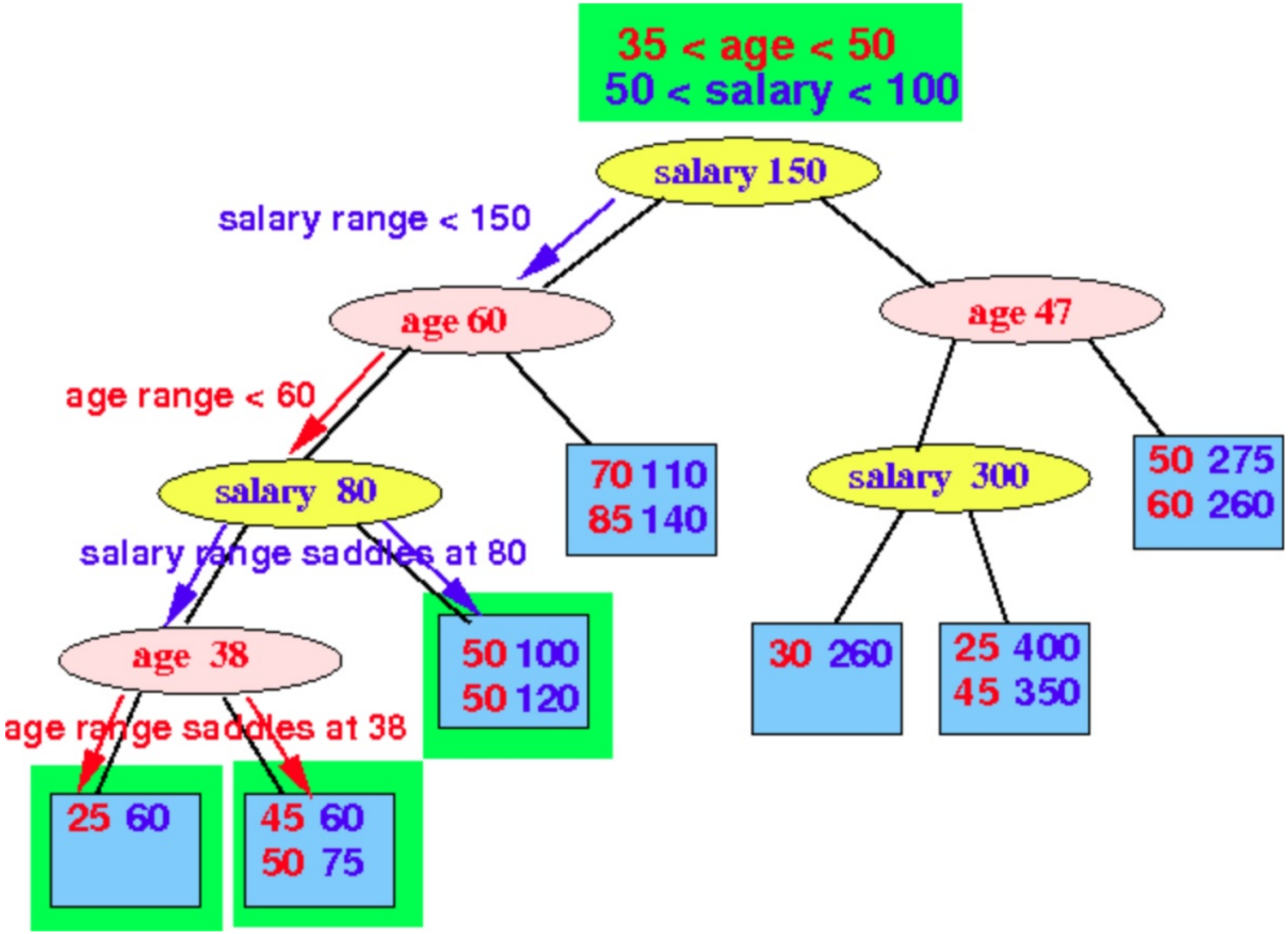
- Find `all person` with `age = 35`



2) Range queries

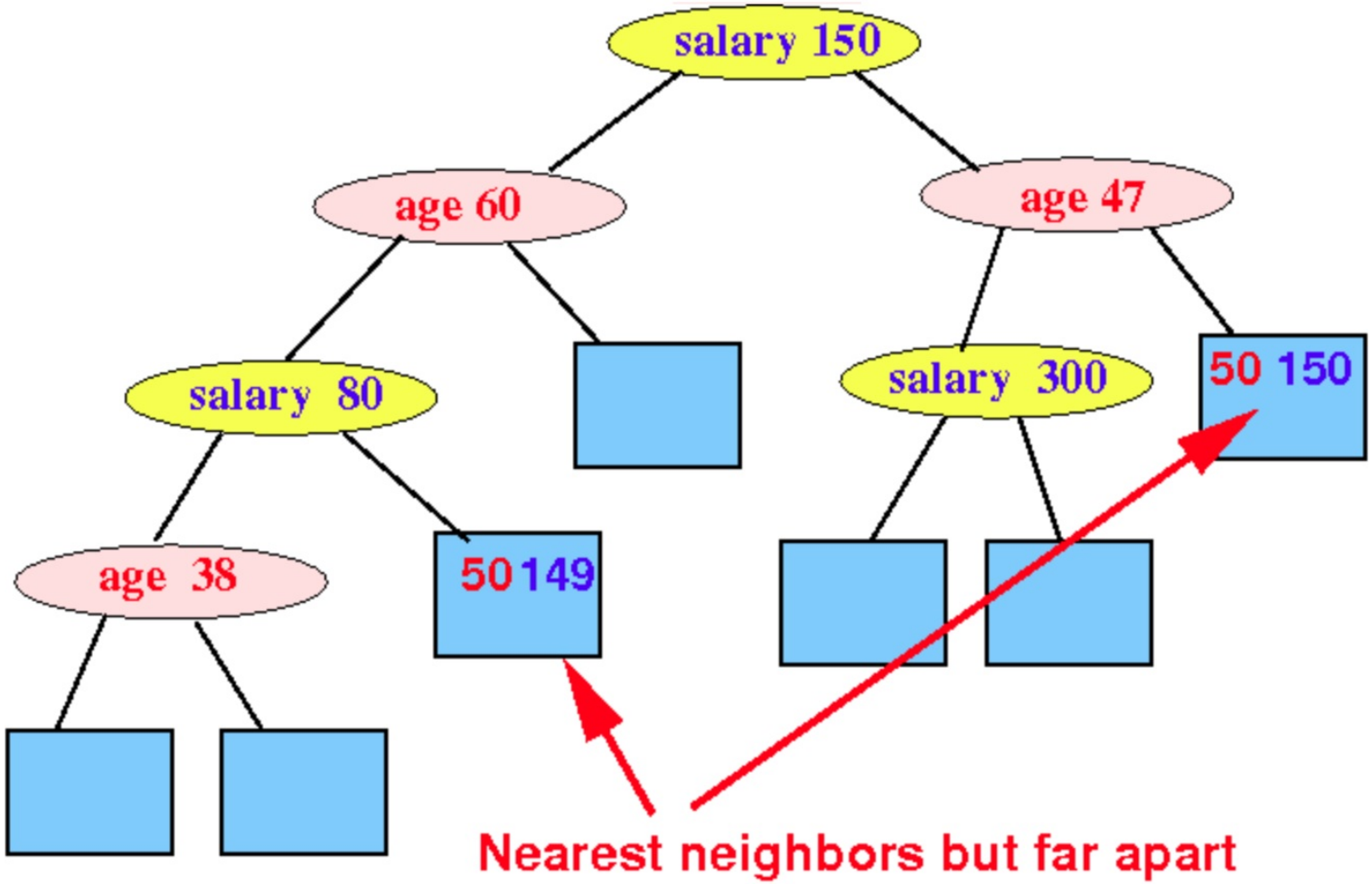
- Search Algorithm
  - For the `search range` is `completely contained` by the `left subtree`, then
    - Take only the `left branch` of the `subtree` for the `search value`
  - For the `search range` is `completely contained` by the `right subtree`, then
    - Take only the `right branch` of the `subtree` for the `search value`
  - Otherwise search both subtrees

2) Range queries: Example



3) Nearest neighbor queries

- Not easy to find the `nearest neighbor` using a `kd-tree index`



- It requires up and down traversal/search in the `kd-tree`

4) Where-am-I queries

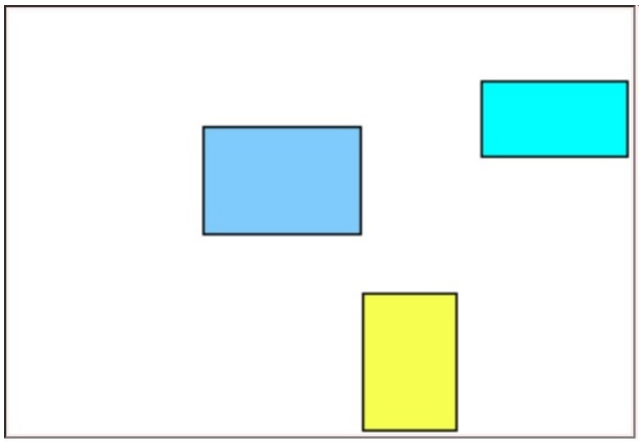
- Not applicable
  - `kd-tree` can only stores `points`
  - Cannot store `objects`

Multi-dimensional index structures

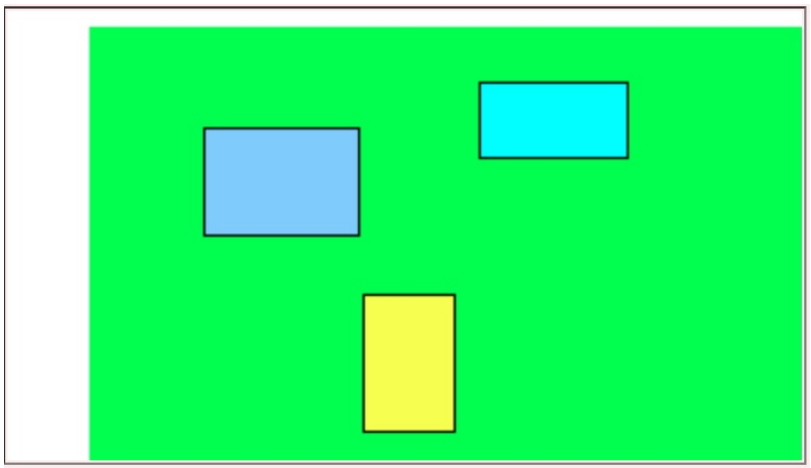
- Hash like structures
  - Grid files
  - Partitioned Hash functions
- Tree like structures
  - Multiple key indexes
  - kd-trees
  - Quad trees
  - R-trees

The R-tree (Region-tree)

- Bounding Box
  - a rectangle that contains a group of objects
- Example: given a group of objects

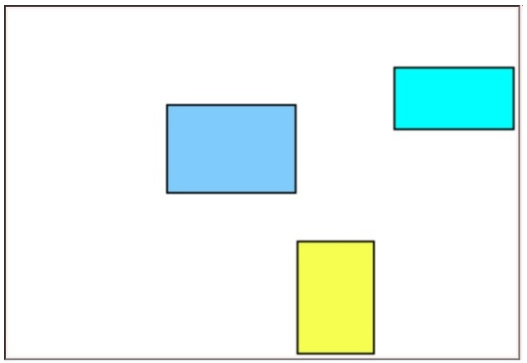


- The Bounding Box for this group of objects

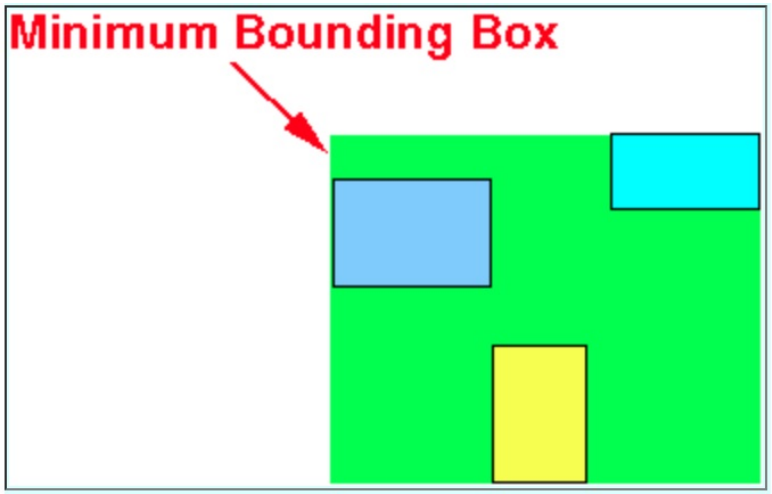


The R-tree (Region-tree)

- Minimum Bounding Box (MBB)
  - the smallest rectangle that contains a group of objects
- Example: given a group of objects

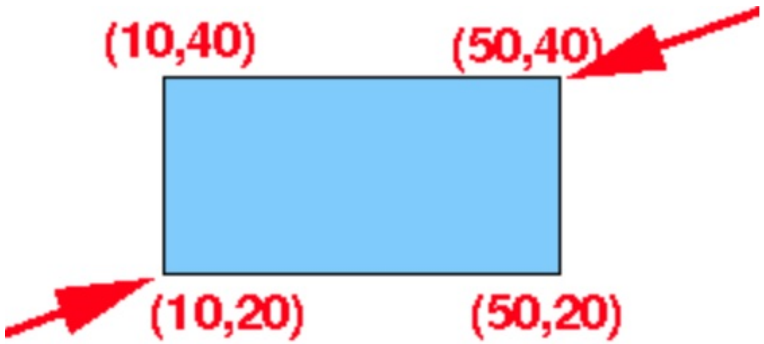


- The Minimum Bounding Box for this group of objects



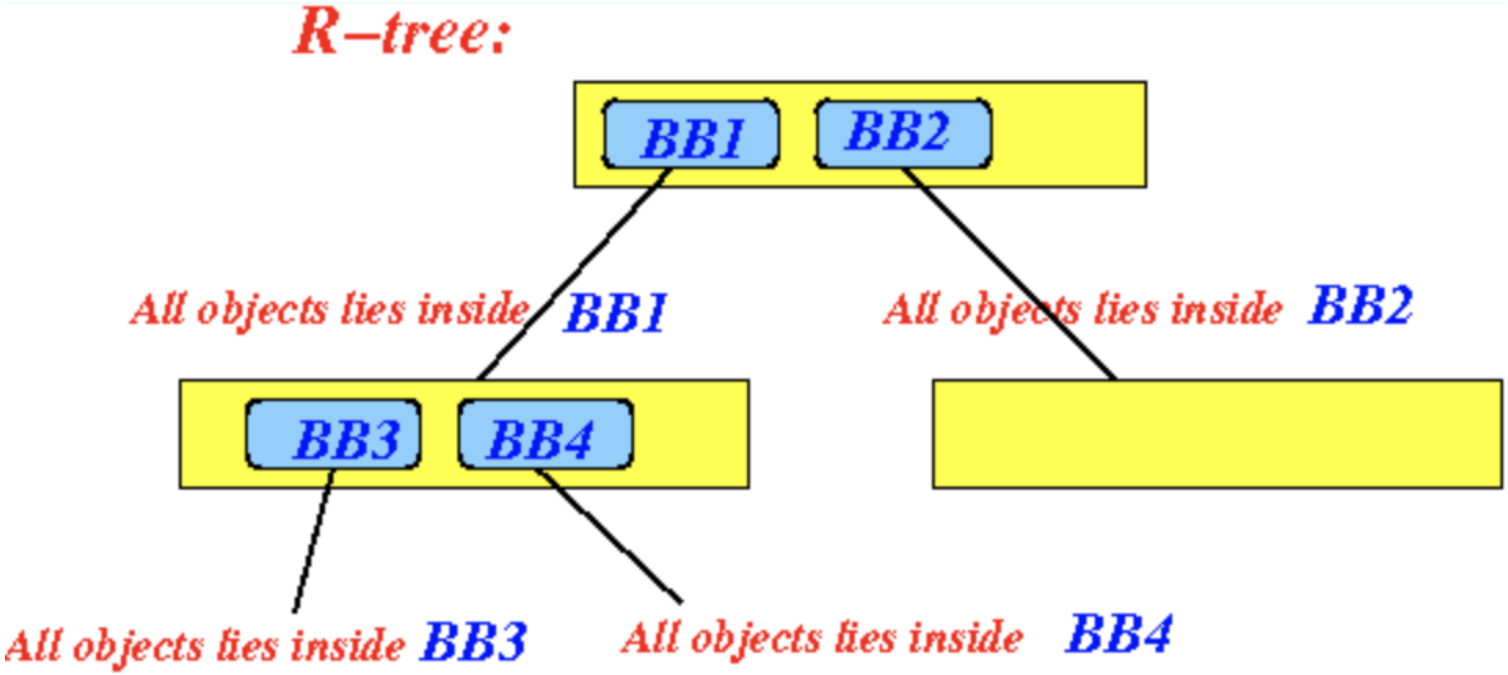
The R-tree (Region-tree)

- **Note:** A `rectangle` can be represented as follows
  - coordinate of the lower left corner
  - coordinate of the upper right corner
- Example: `Rectangle:` `((10,20), (50,40))`



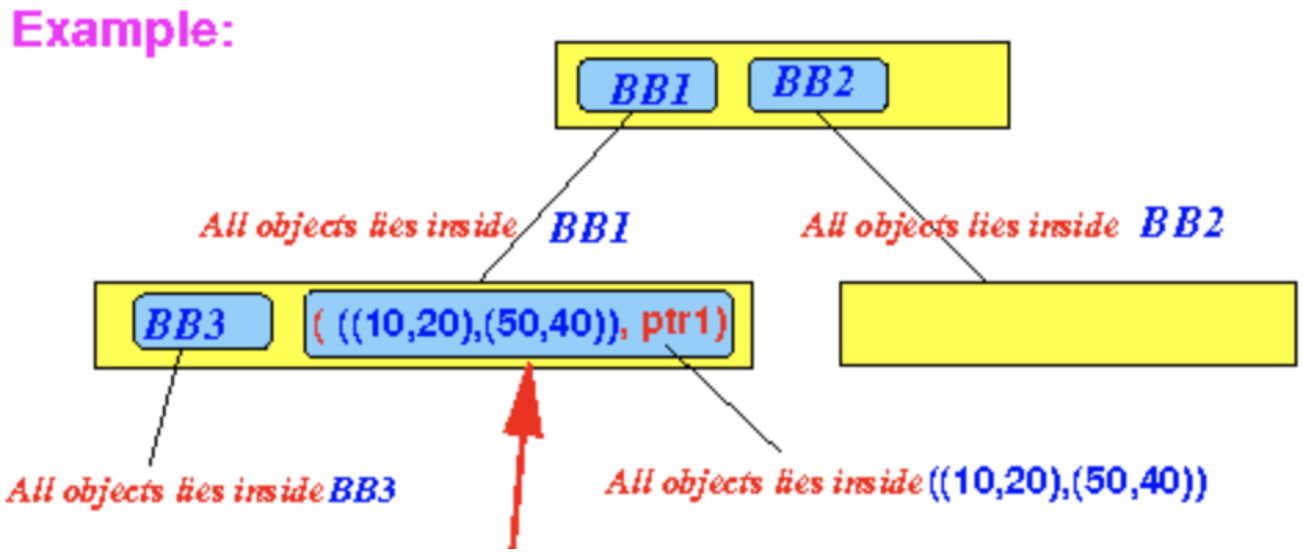
The R-tree (Region-tree)

- **R-Tree:** an index tree-structure derived from the B$^+$-tree that uses bounding boxes as search keys
- Example



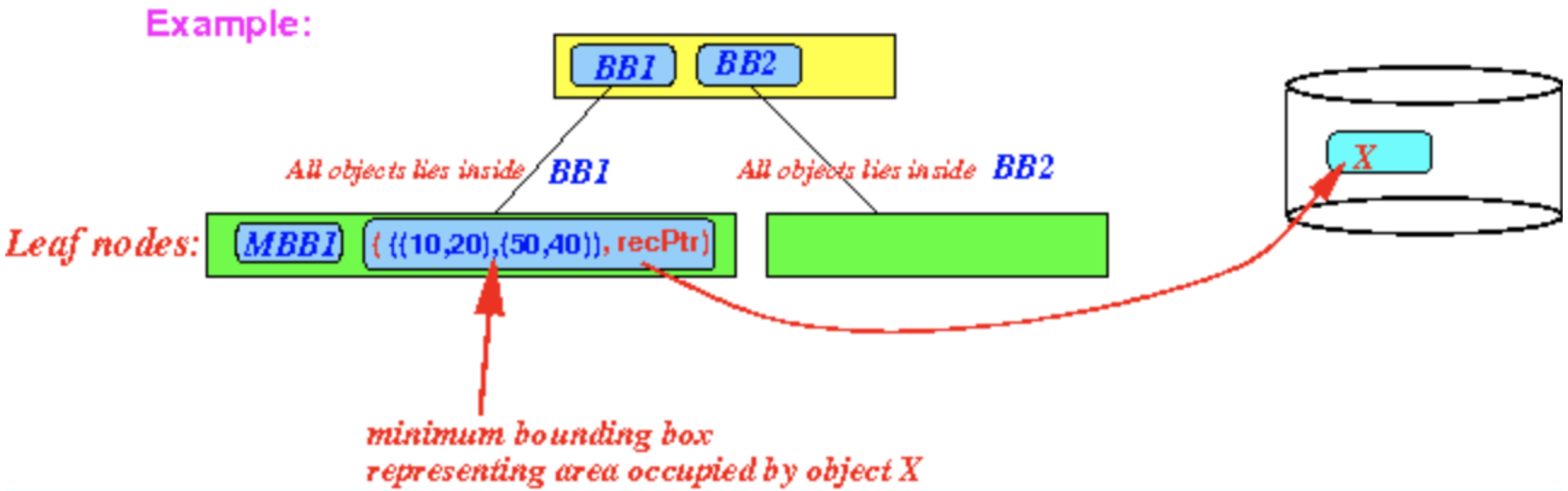
The R-tree (Region-tree)

- The `internal nodes` contains a number of entries of the following format
  - **(bounding box, child node pointer)**
  - Example: $\Big(\big((10,20),(50,40)\big),\texttt{ptr1}\Big)$



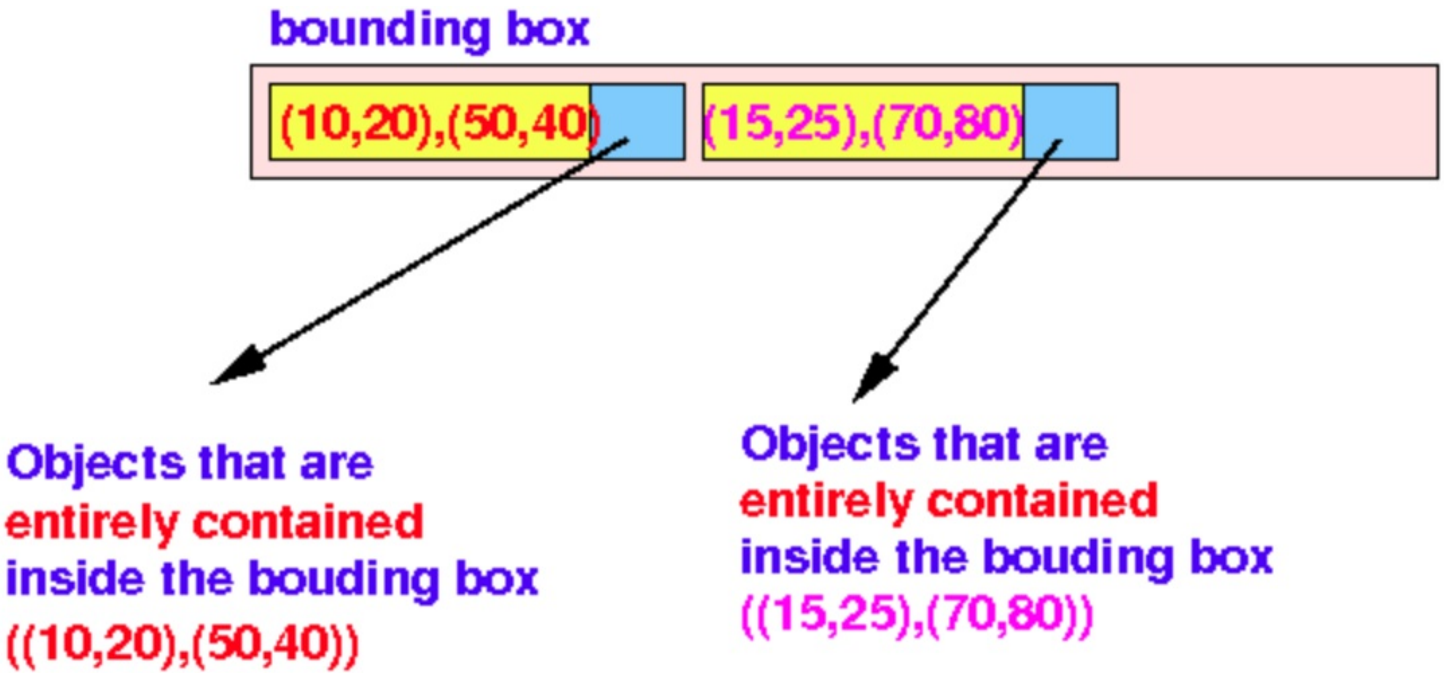
The R-tree (Region-tree)

- The `leaf nodes` contains a number of entries of the following format:
  - **(min bounding box, object pointer)**
  - Example: $\Big(\big((10,20),(50,40)\big),\texttt{house-ptr}\Big)$



Search property of a `R-tree`

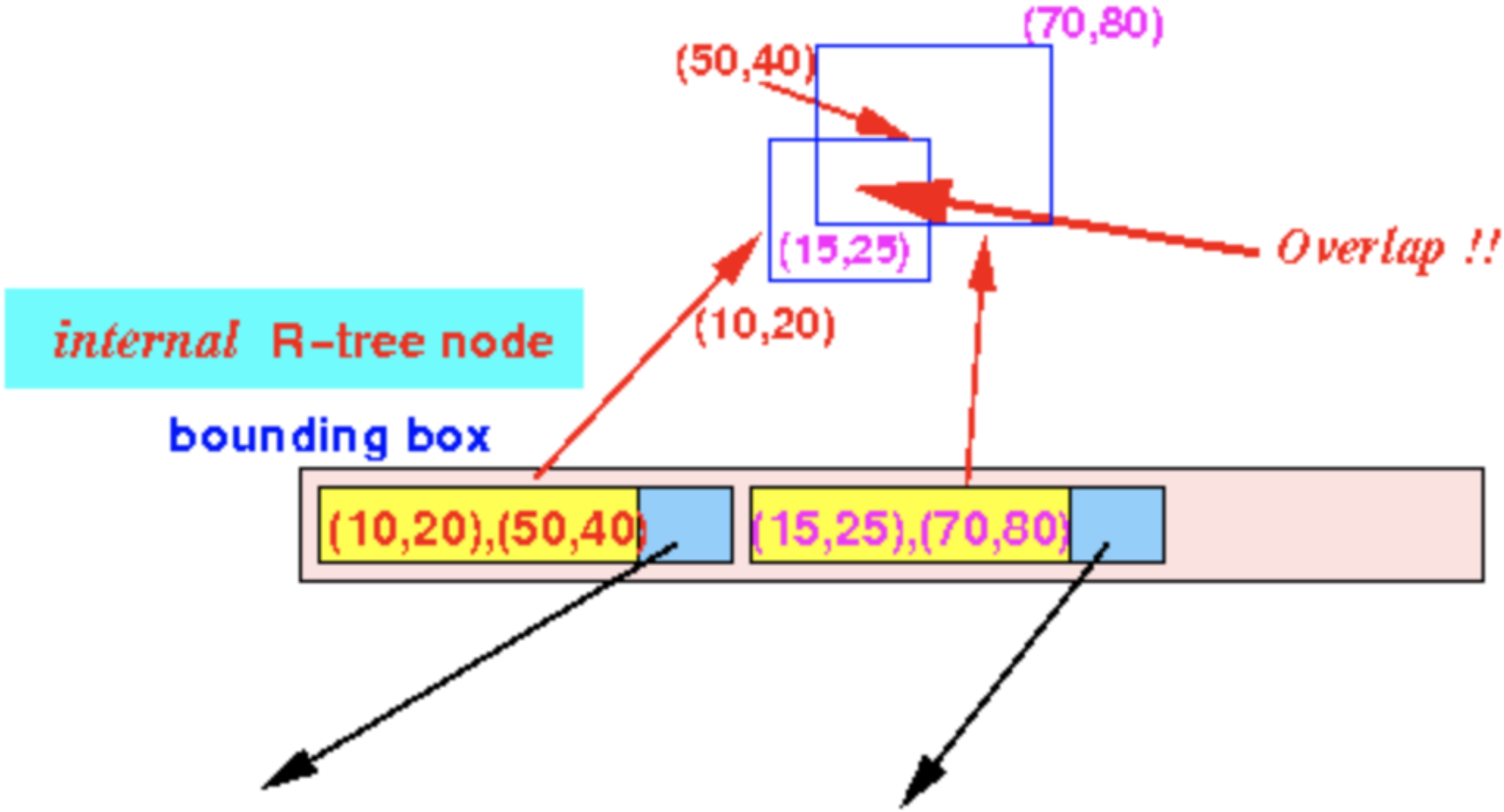
- The sub-tree indexed by an `internal node` of the `R-tree` has the following structure:



- i.e., the `subtree` indexed by the `bounding box` will contain
  - Only objects that is contained within the given `bounding box`

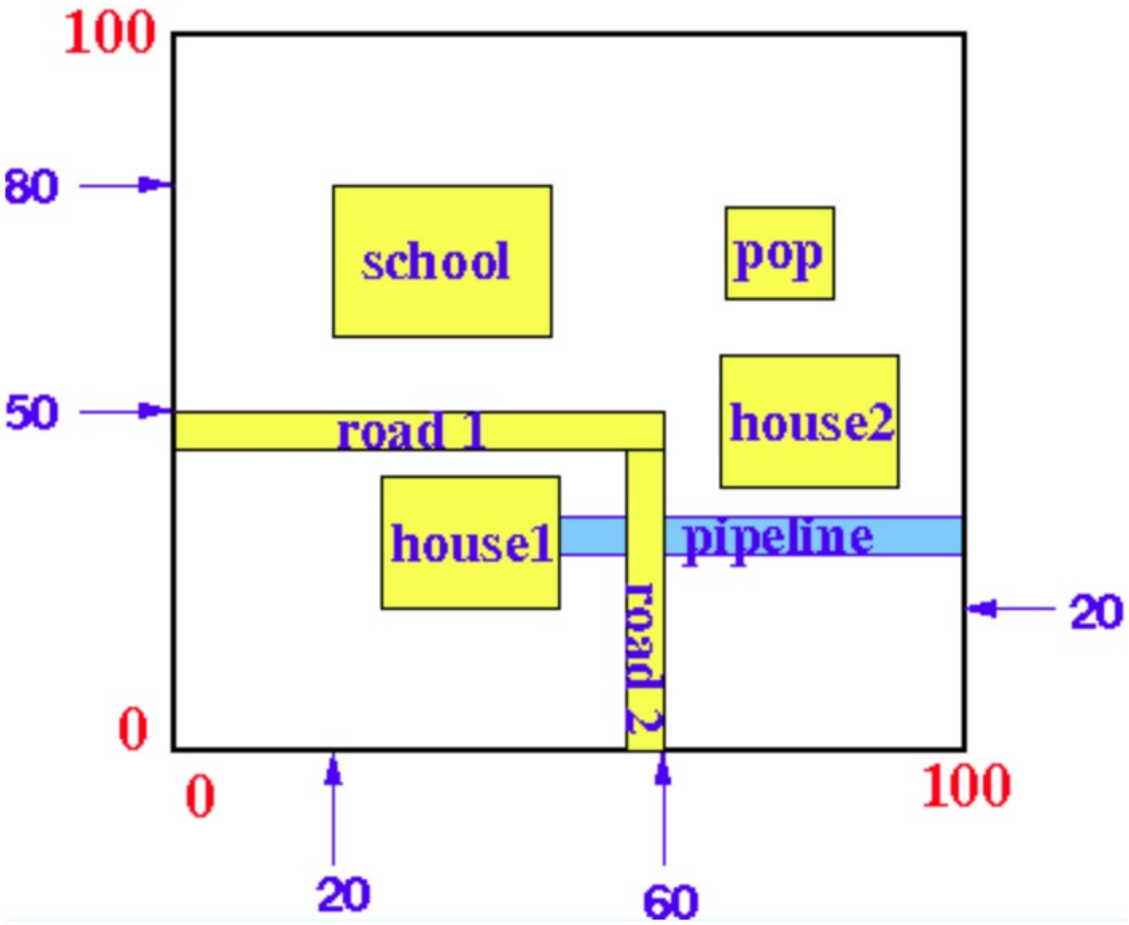
Important note

- `Bounding boxes` in a `R-tree` node can *overlap*



R-tree: Example

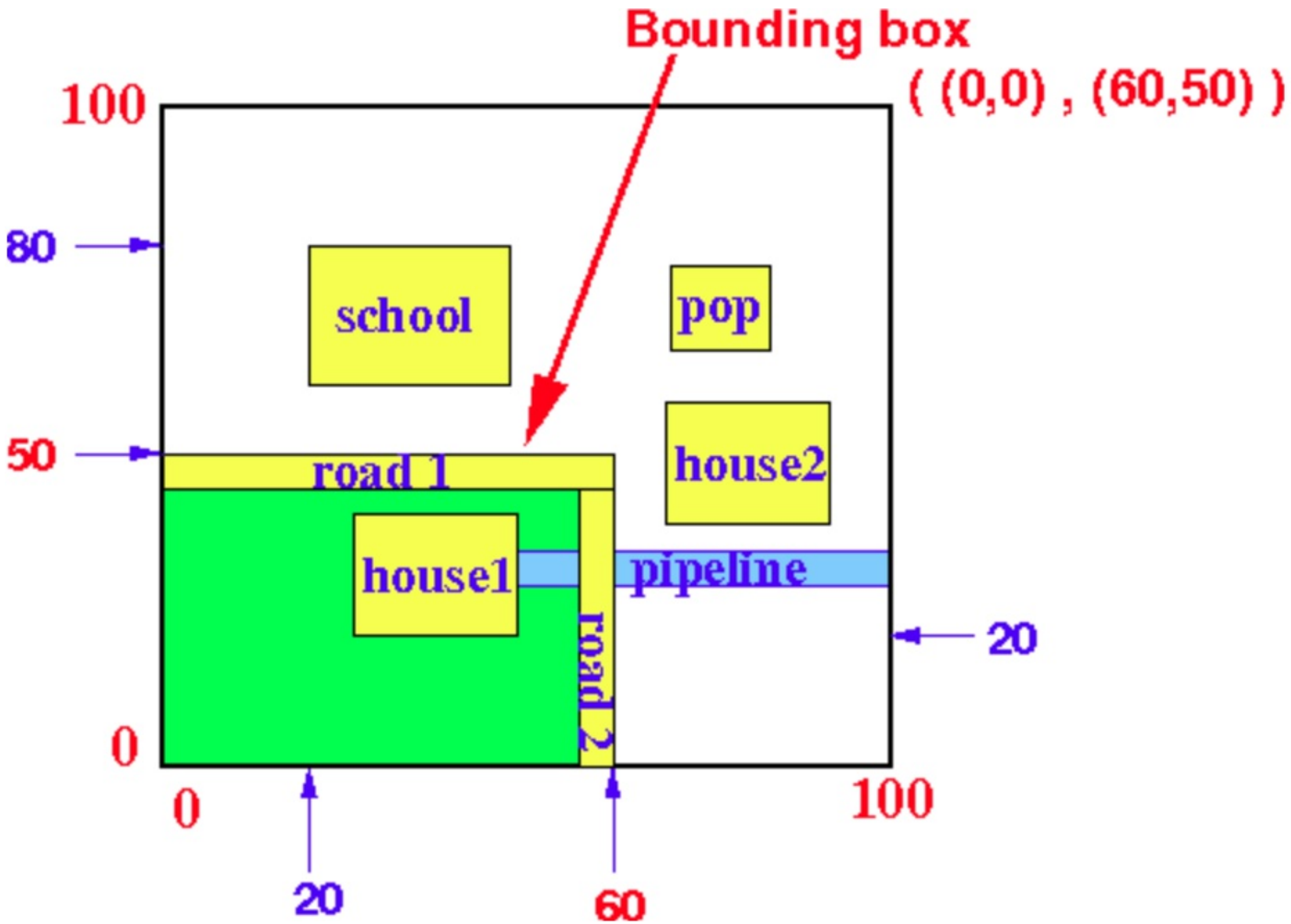
- Objects that we want to represent



- There are 7 objects
  - school, pop (point of presence), house1, house2, road1 road2, pipeline

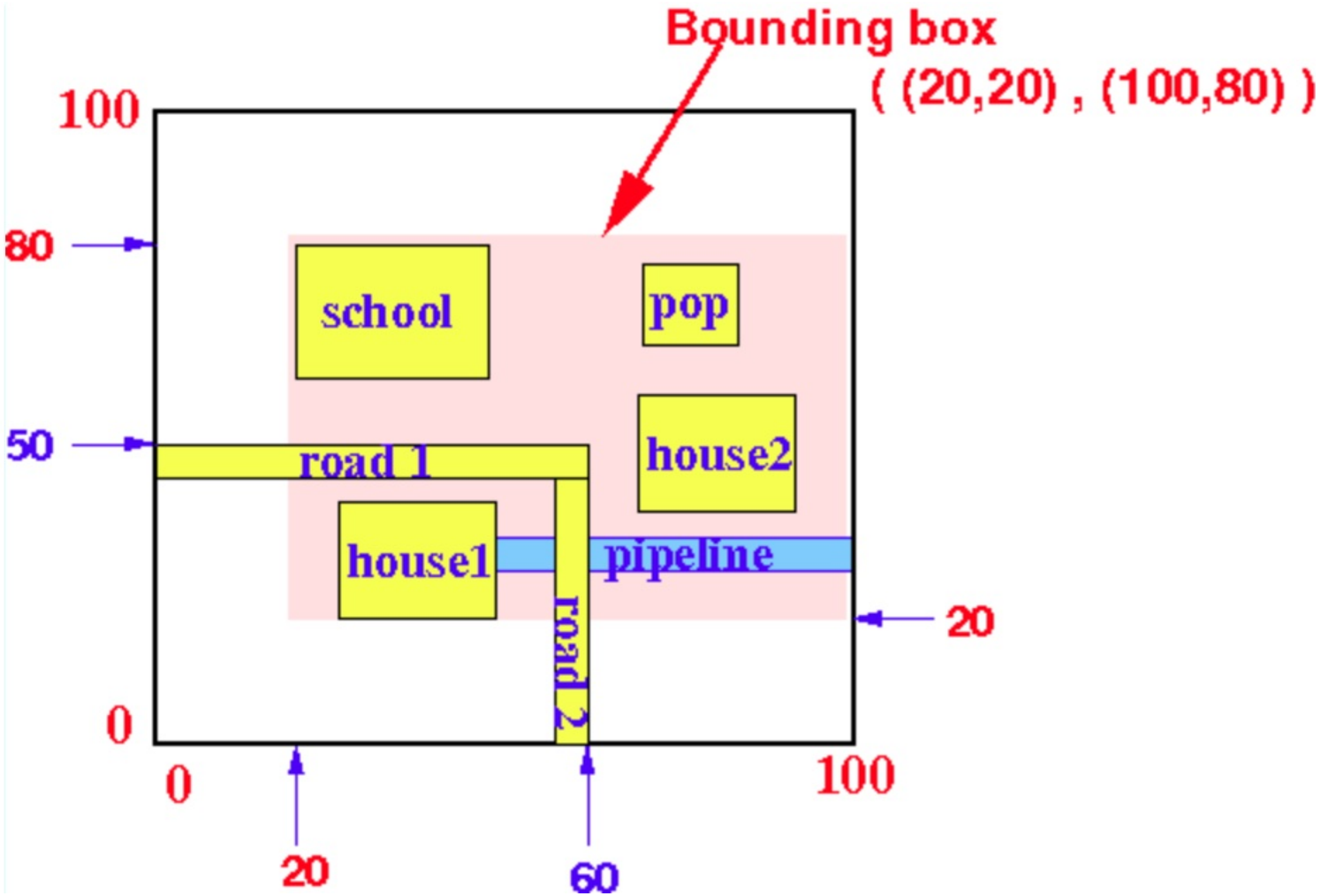
R-tree: Example

- The 3 objects house1, road1 and road2 are completely enclosed by the `bounding box` `((0,0),(60,50))`



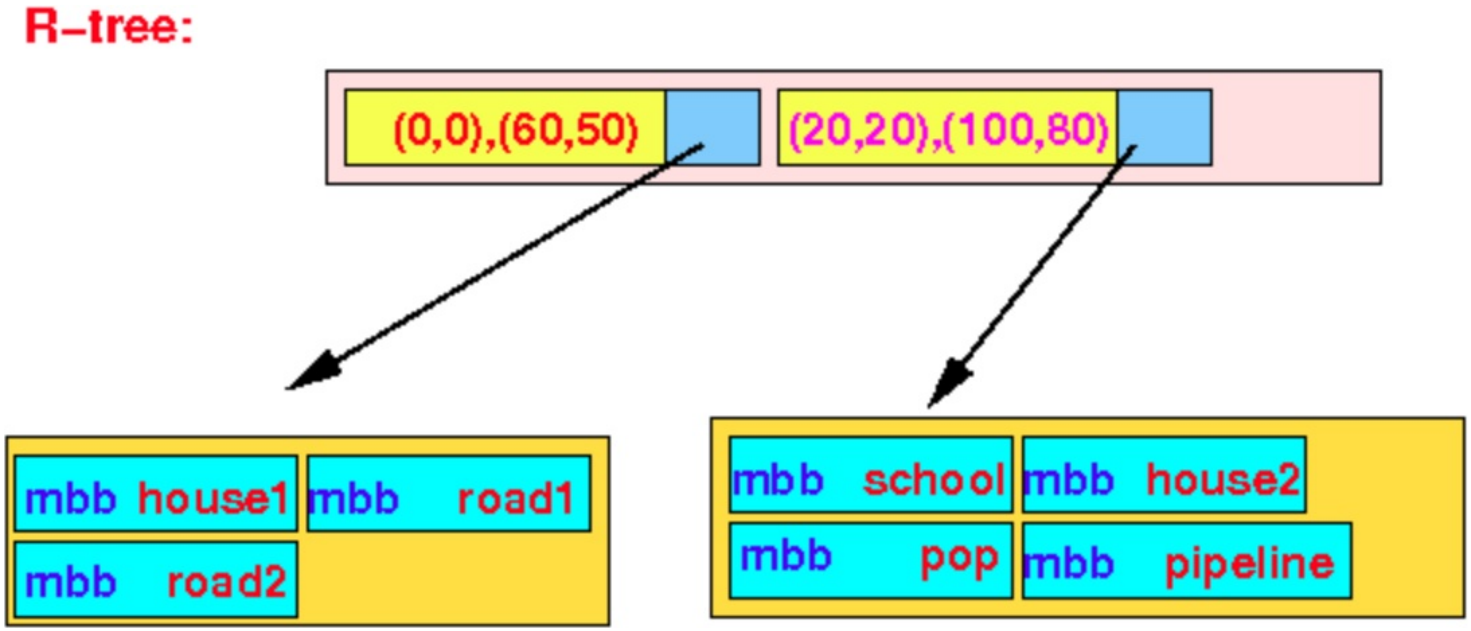
R-tree: Example

- The objects school, pop, house2 and pipeline are completely enclosed by the `bounding box` `((20,20),(100,80))`



R-tree: Example

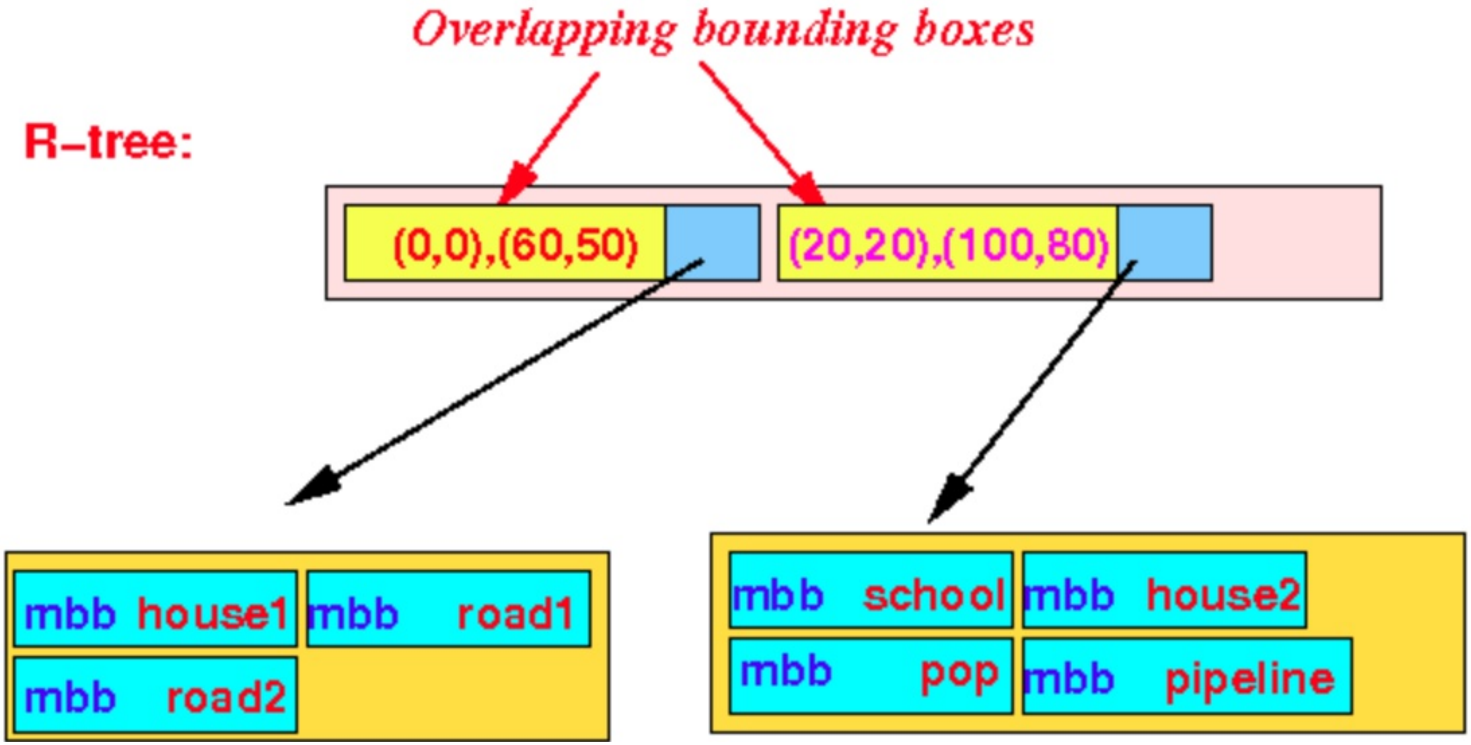
- The R-tree that uses the previous bounding boxes



- The minimum bounding box (mbb) field for different objects are different

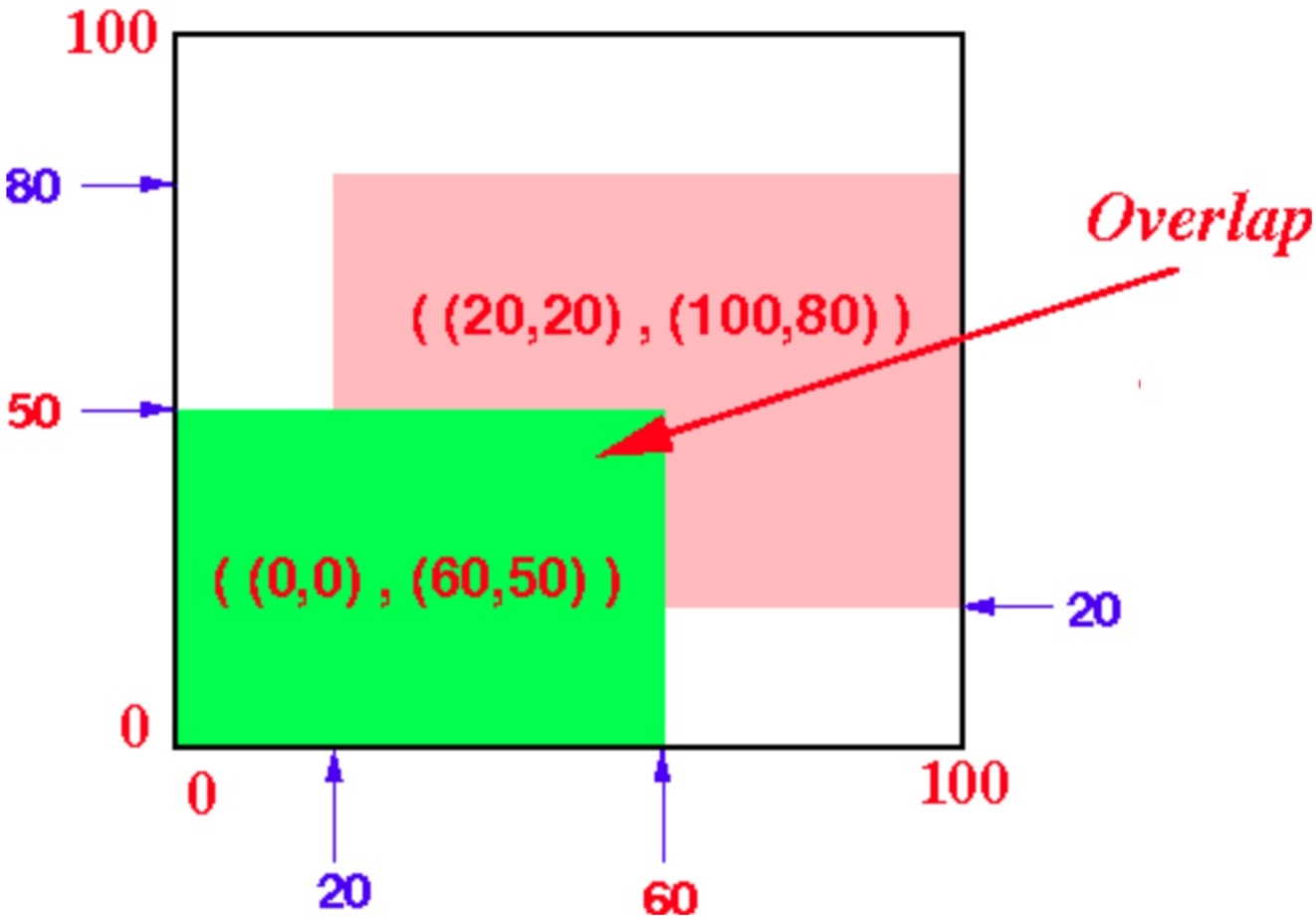
Overlapping Bounding boxes in `R-tree`

- The `bounding boxes` used in the `internal R-tree nodes` can *overlap*
- Example



Overlapping Bounding boxes in `R-tree`

- You can see the `overlap` clearly



Search algorithm in an R-tree

- The search algorithm in an **R-tree** is based on the **depth-first search tree traversal** algorithm
- A branch of the **R-tree** will be search only if the object that you want to find is located within the corresponding **bounding box** (search key)
- Searching problem for a **Point(x,y)**
  - Given a point (x,y)
  - Find the object(s) in the R-tree that contains the **point (x,y)**
- Search Algorithm for a **Point(x,y)**
  - The **search algorithm** is recursive
  - The search starts at the **root node** of the **R-tree**

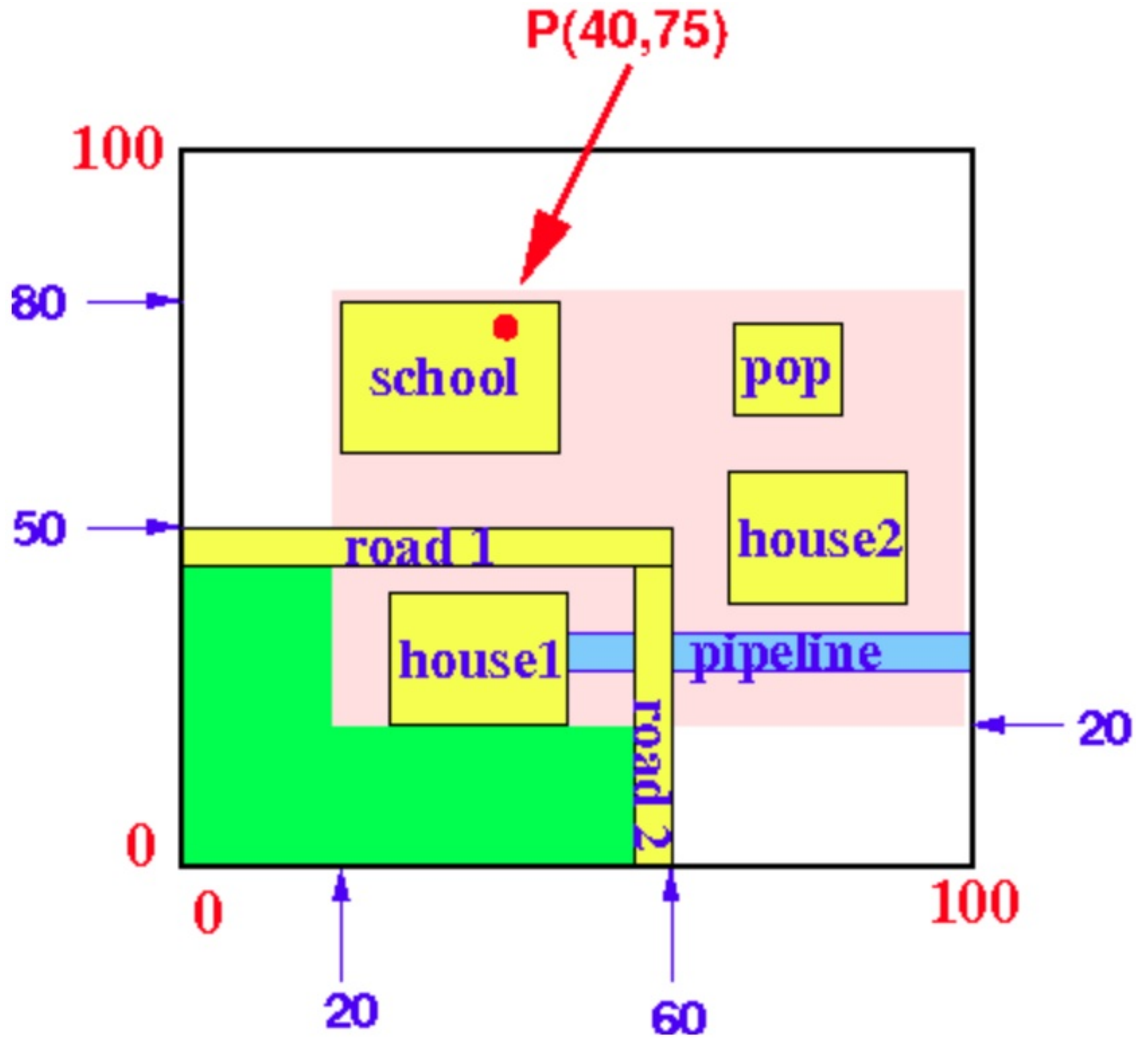
Search algorithm for a point `P(x,y)`

Algorithm 2 Lookup$\big((x, y)$, n, result$\big)$

```
1:  // n = current node of the search in the R-tree
2:  if ( n = internal node ) then
3:     for each entry (BB, childptr) in internal node n ) do
4:        // Look in subtree if (x, y) is inside bounding box
5:        if (x, y) ∈ BB then
6:           Lookup((x,y), childptr, result)
7:        end if
8:     end for
9:  else
10:    //n is a leaf node
11:    for ( each object Ob in node n) do
12:       if (x,y) ∈ MBB(Ob) then
13:          Add Ob to result // Object Ob contains point (x, y)
14:       end if
15:    end for
16: end if
```

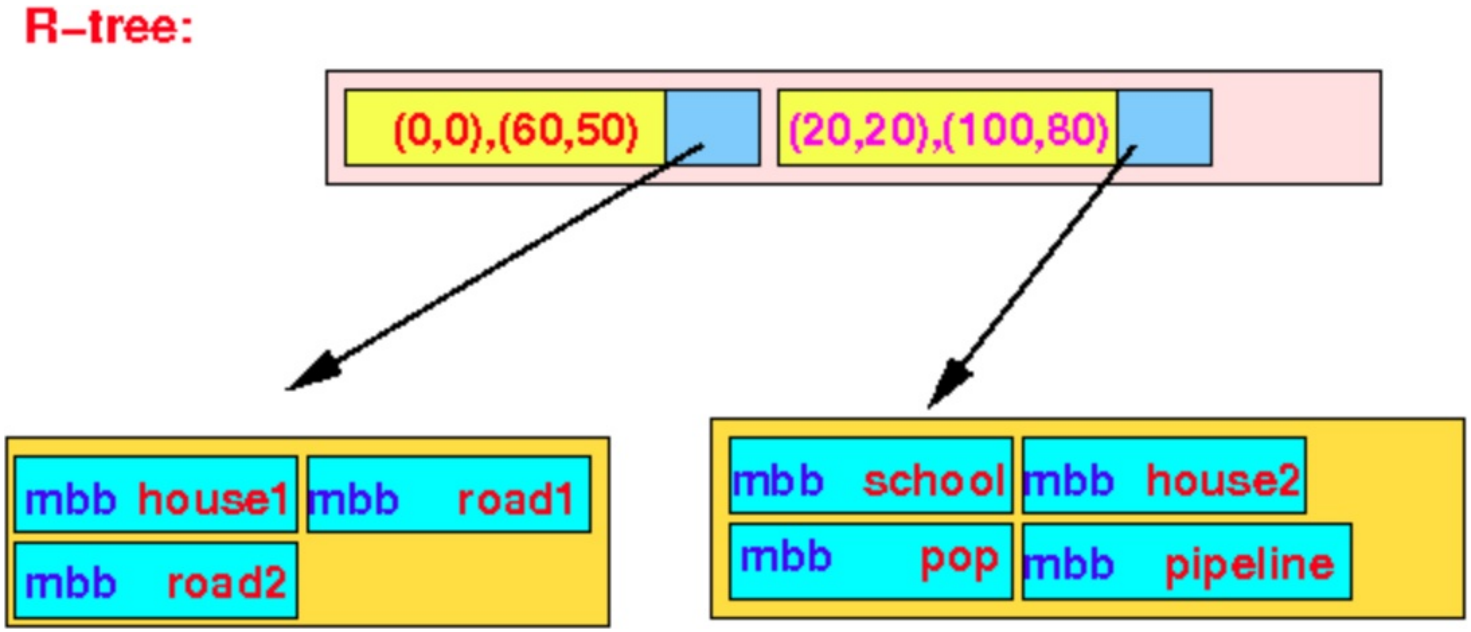
Lookup operation in the `R-tree`: Example

- Find the object(s) that contain the point `P`$(40, 75)$



Lookup operation in the `R-tree`: Example

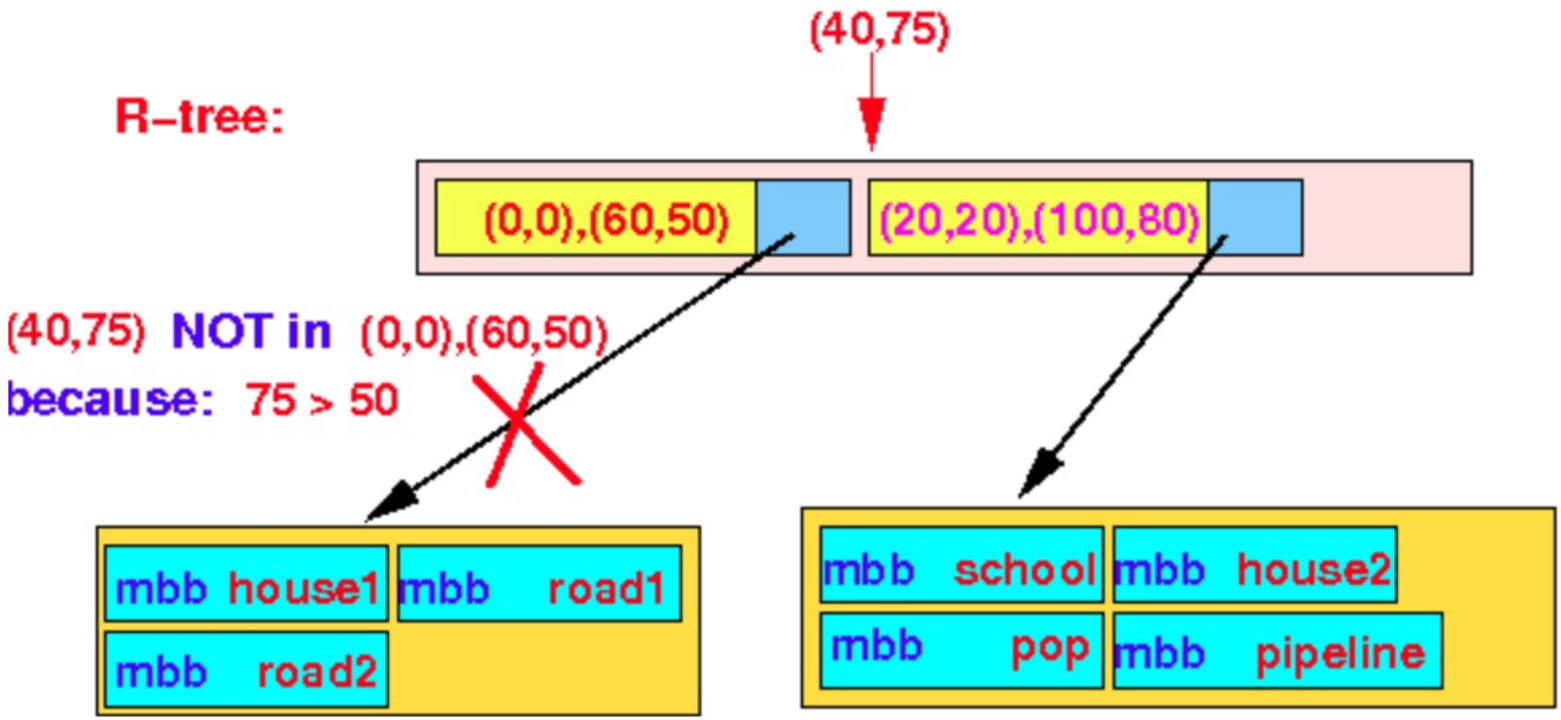
- Execution of the search algorithm:
  - Input:



  - `Point (40,70)`

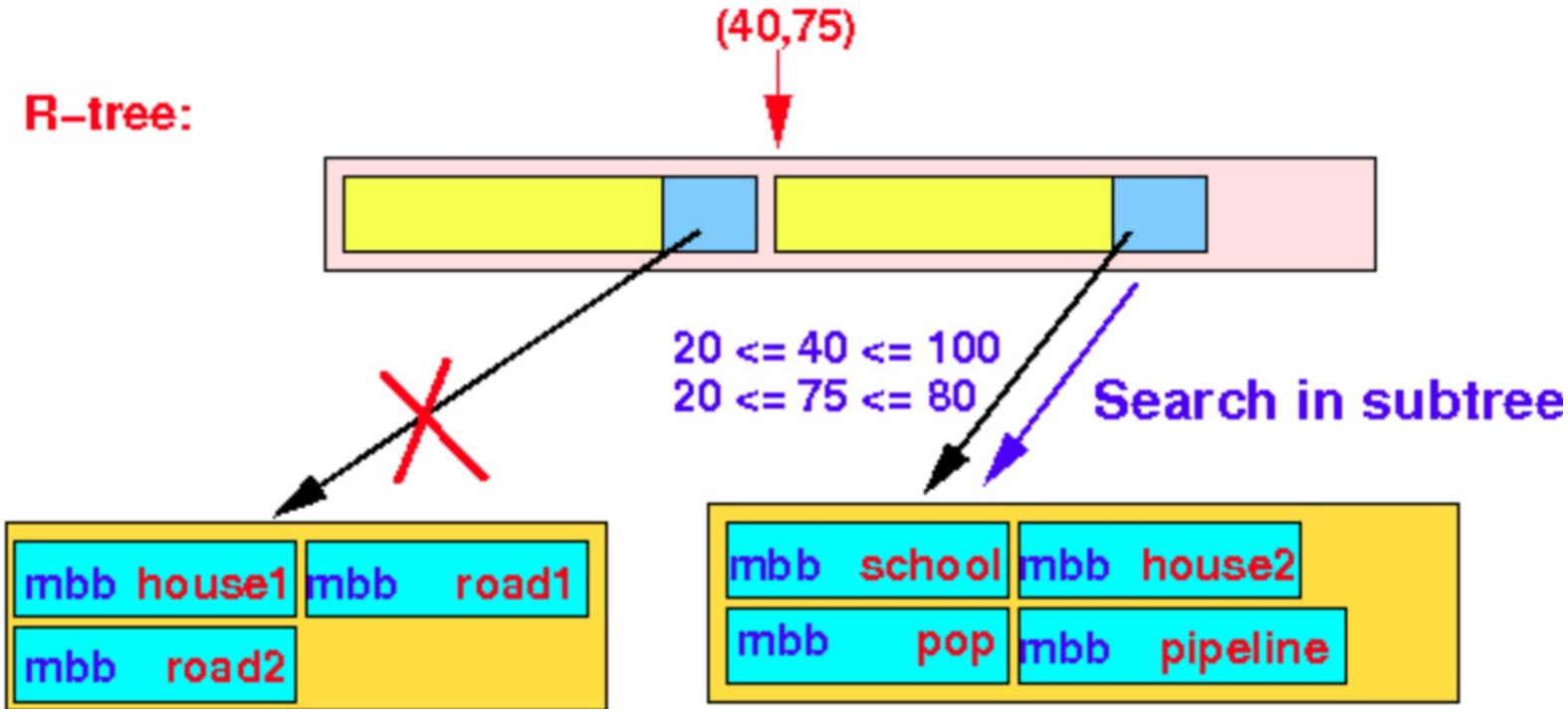
Lookup operation in the `R-tree`: Example

- Execution of the search algorithm:
  - Check in first Bounding Box
    - `point P(40,75)` $\notin$ `bounding box ((0,0),(60,50))`
  - Skip the first subtree



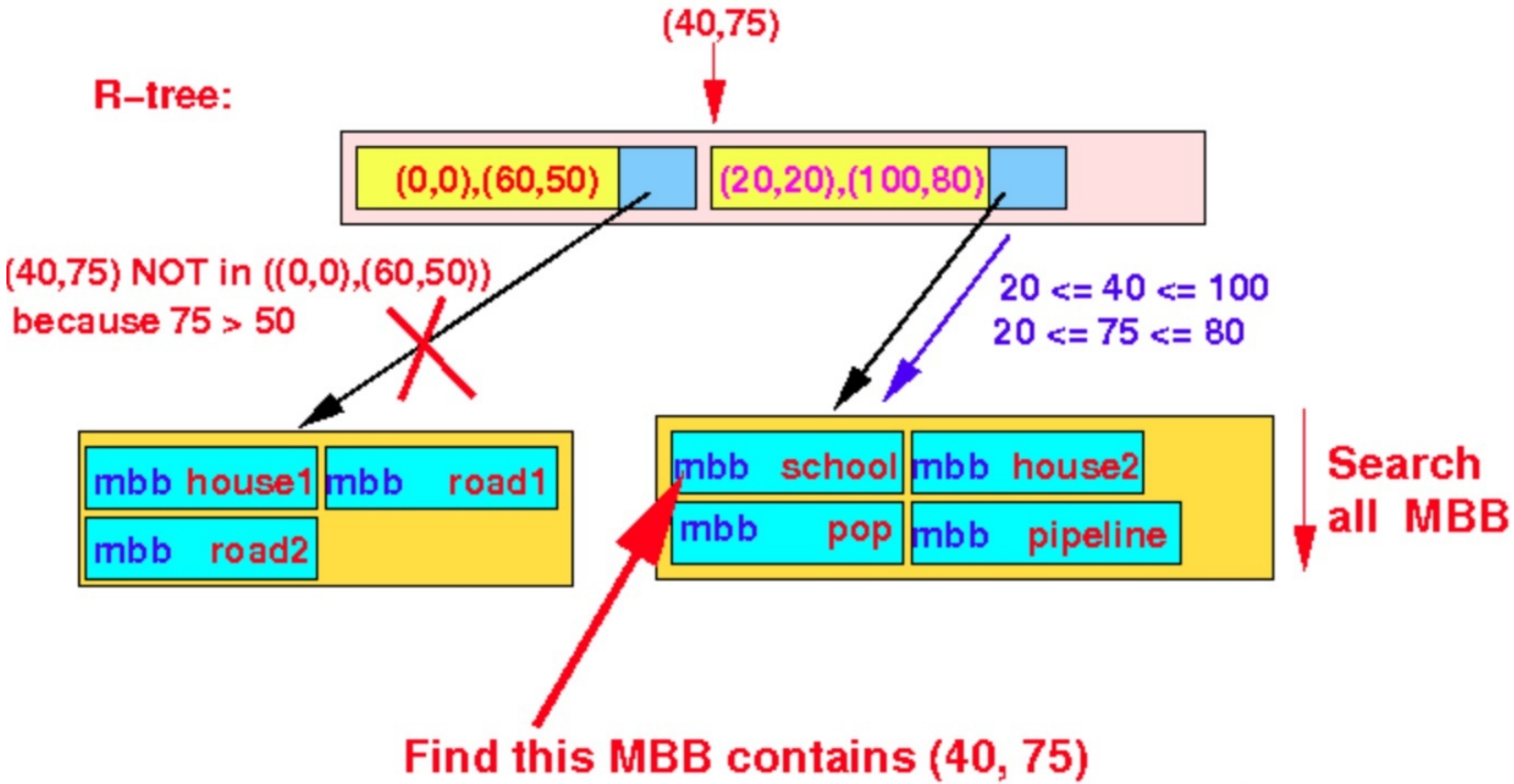
Lookup operation in the `R-tree`: Example

- Execution of the search algorithm:
  - Check in second Bounding Box
    - `point P(40,75)` $\in$ `bounding box ((20,20),(100,80))`
  - The search algorithm will recurse and search the 2^{nd} subtree



Lookup operation in the `R-tree`: Example

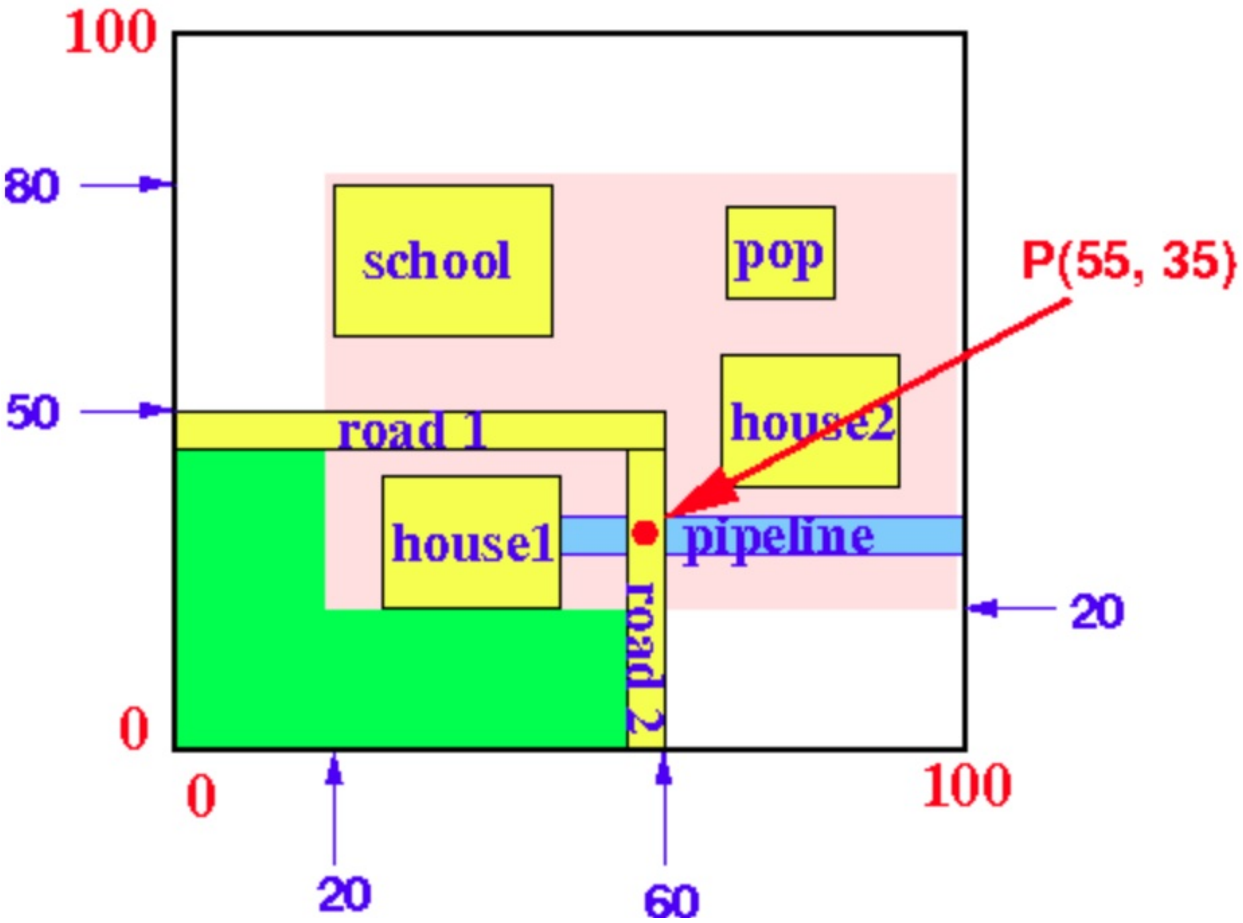
- Execution of the search algorithm:
  - We arrived at a `leaf node`, so we search the `MBBs` in the `leaf node`



  - We find that the `school object` contains the `point`

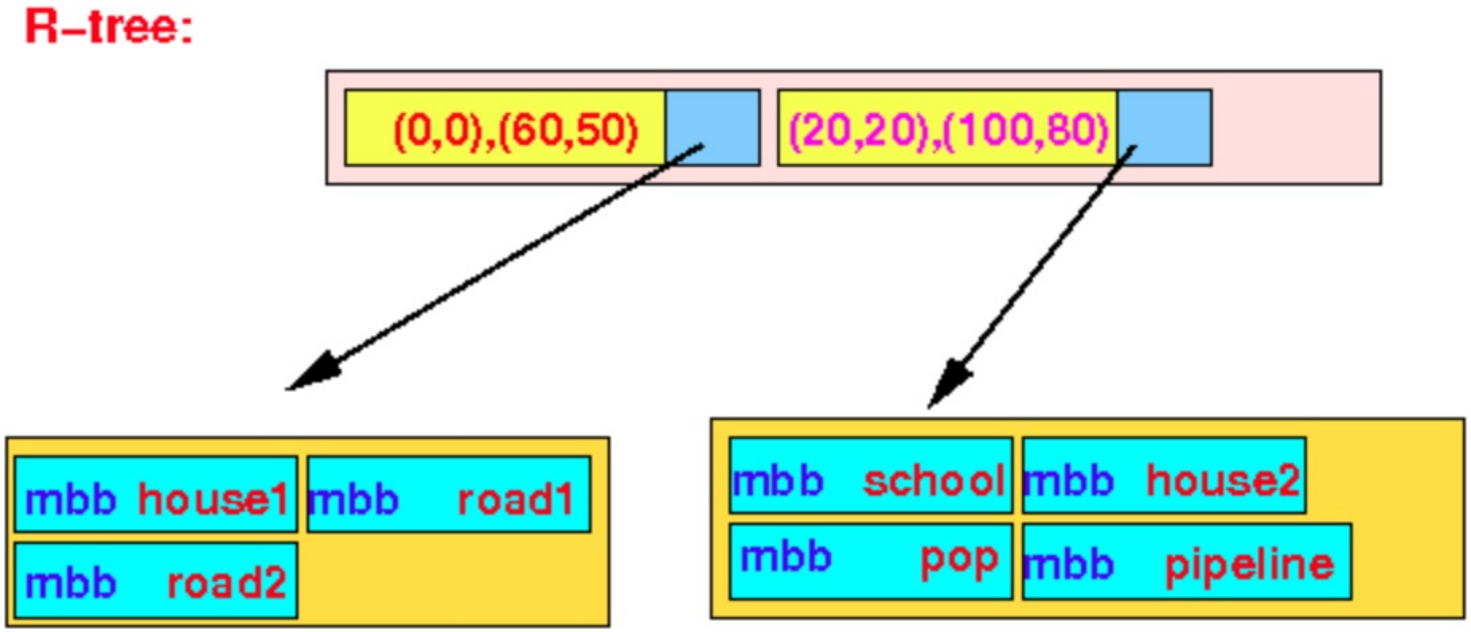
Lookup operation in the `R-tree`: Example 2

- Find the object(s) that contain the `point P(55,35)`



Lookup operation in the `R-tree`: Example

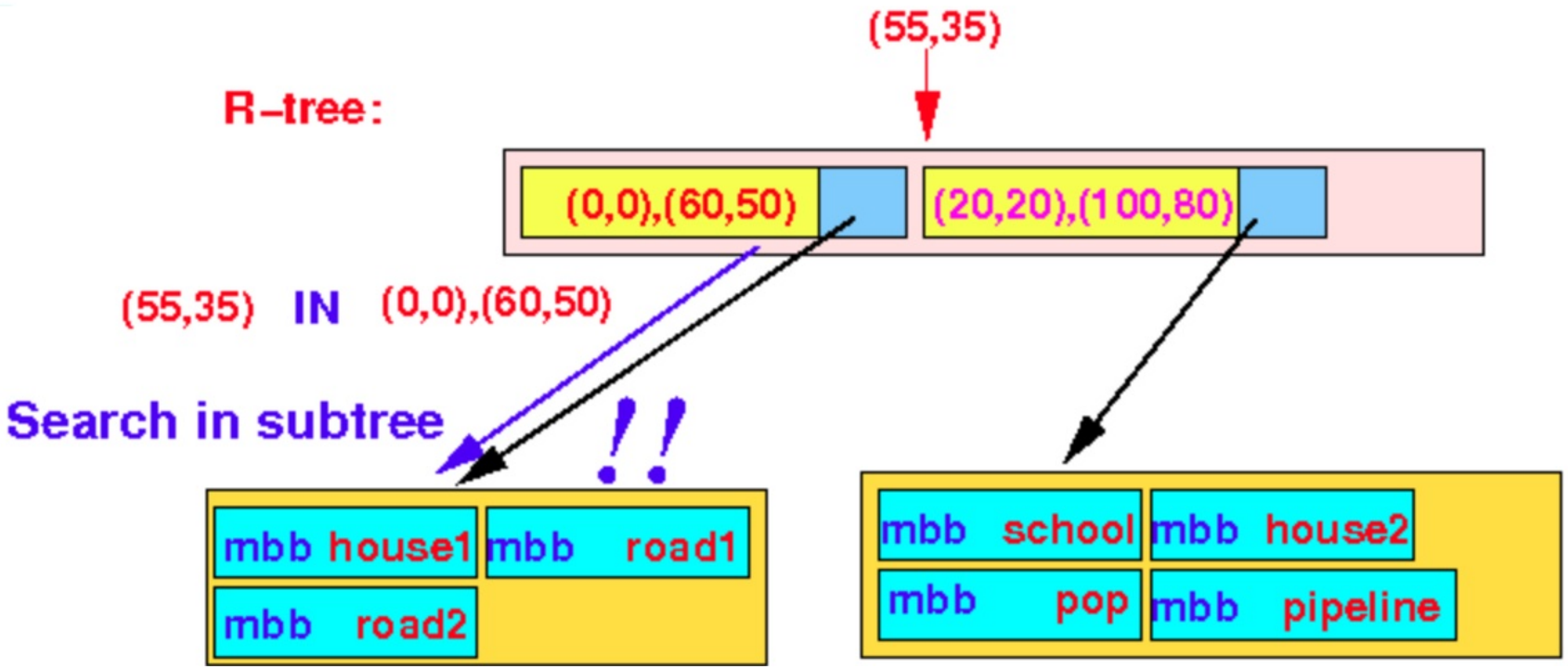
- Execution of the search algorithm:
  - Input:



  - `Point (55,35)`

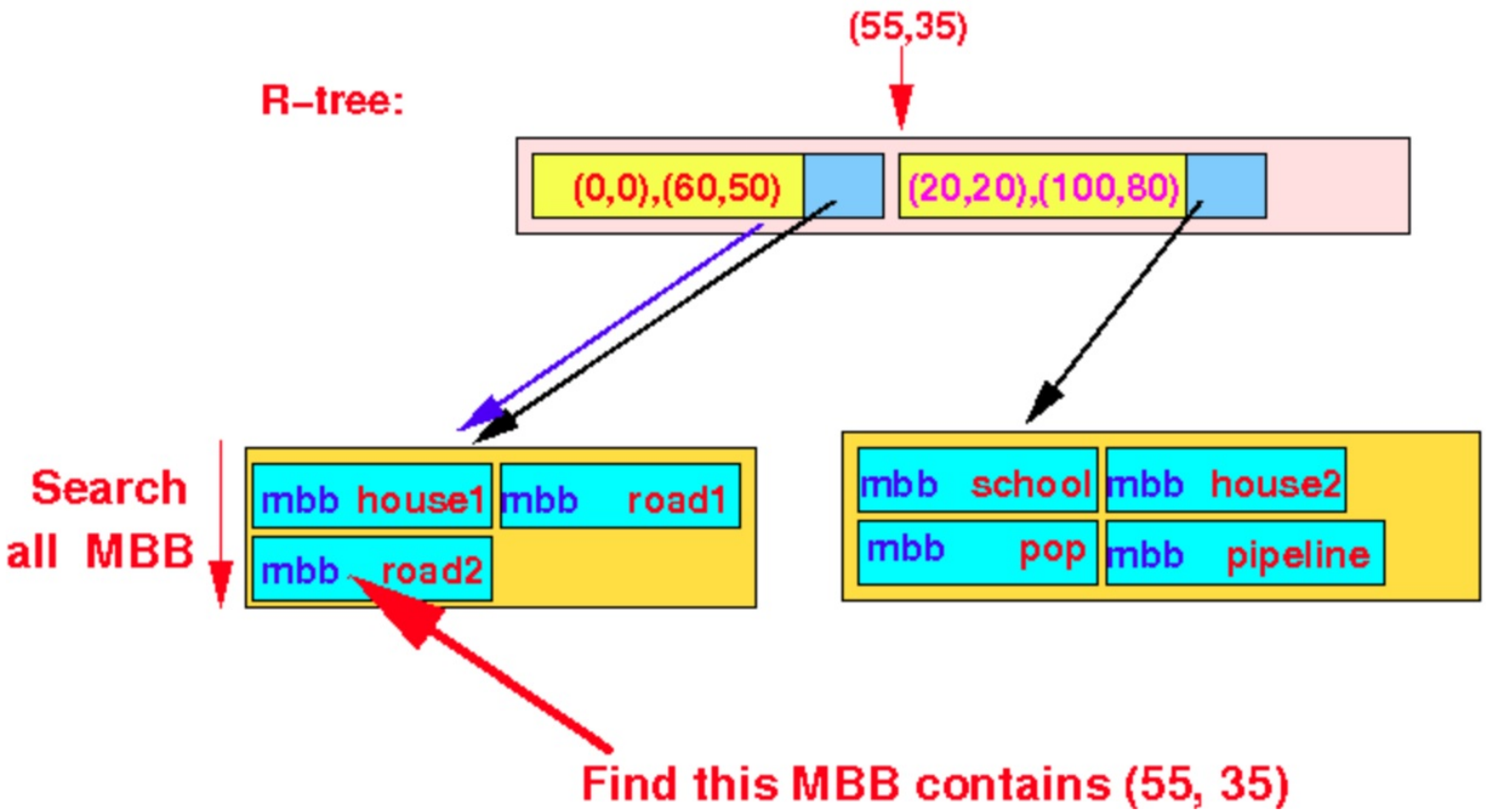
Lookup operation in the `R-tree`: Example 2

- Execution of the search algorithm:
  - Check in first Bounding Box
    - `point P(55,35)` $\in$ `bounding box ((0,0),(60,50))`
  - The search algorithm will recurse and search the 1^{st} subtree



Lookup operation in the `R-tree`: Example 2

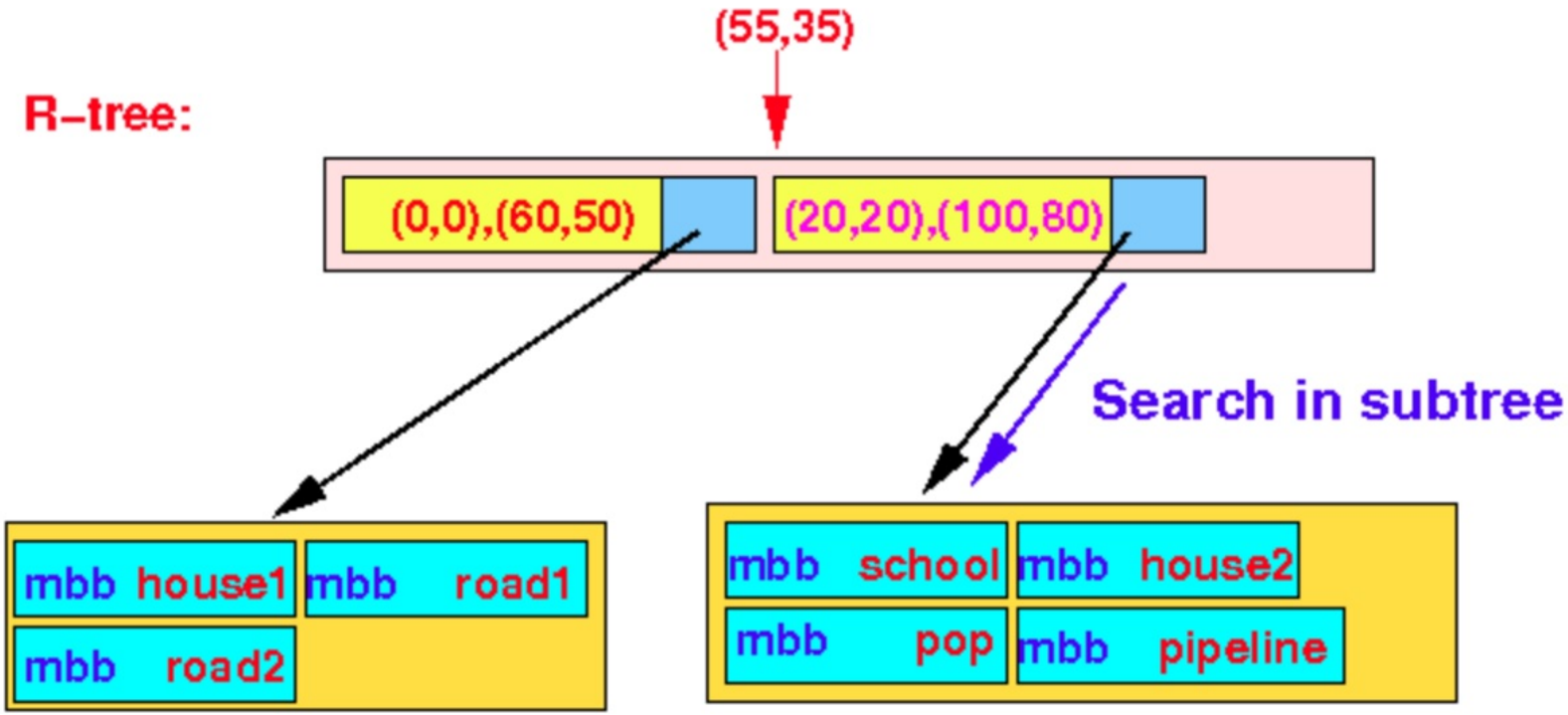
- Execution of the search algorithm:
  - We arrived at a `leaf node`, so we search the `MBBs` in the `leaf node`



  - We find that the `road2 object` contains the `point`

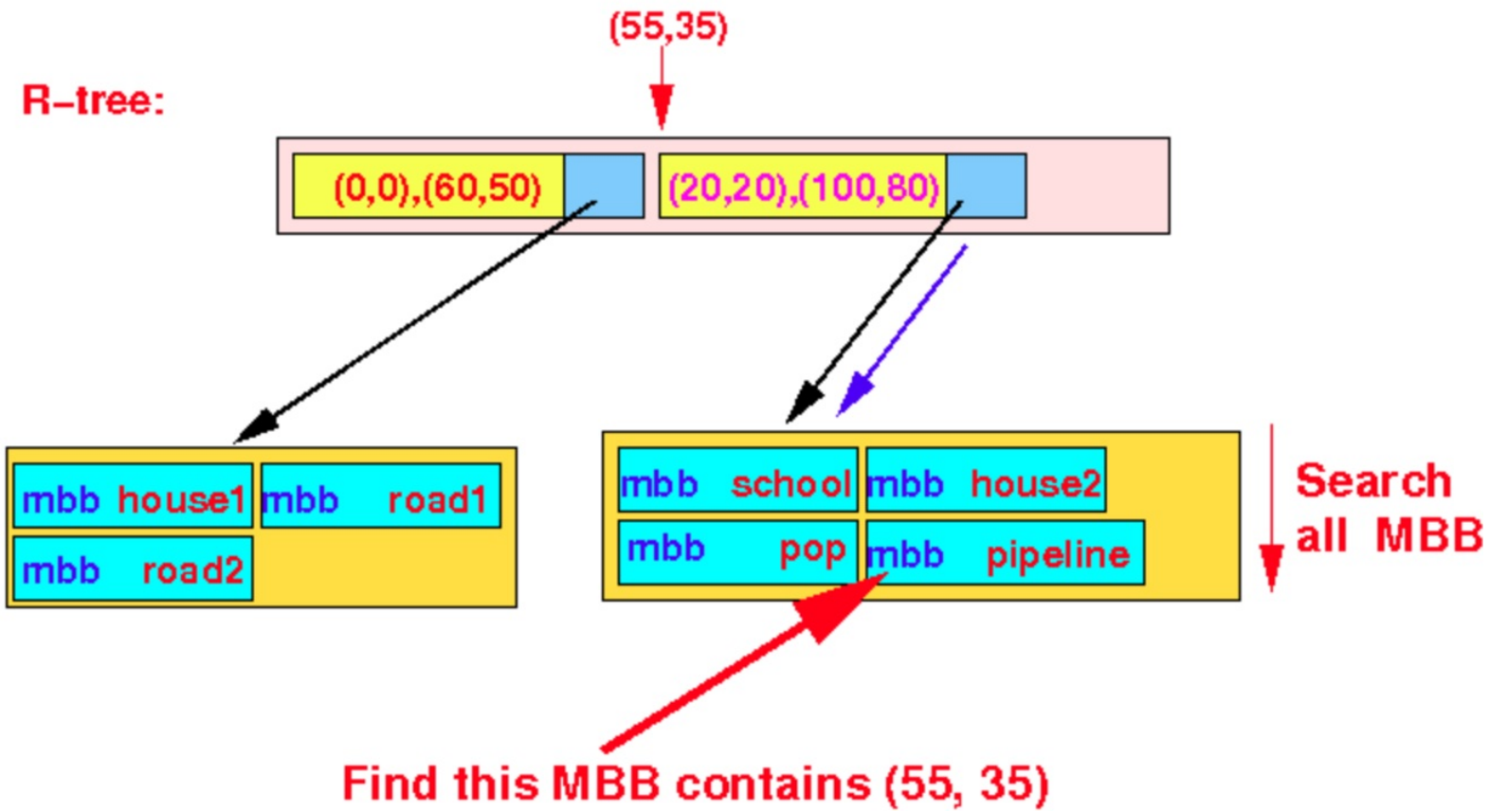
Lookup operation in the `R-tree`: Example 2

- Execution of the search algorithm:
  - Check in second Bounding Box
    - `point P(55,35) ∈ bounding box ((20,20),(100,80))`
  - The search algorithm will also recurse and search the 2^{nd} subtree



Lookup operation in the `R-tree`: Example 2

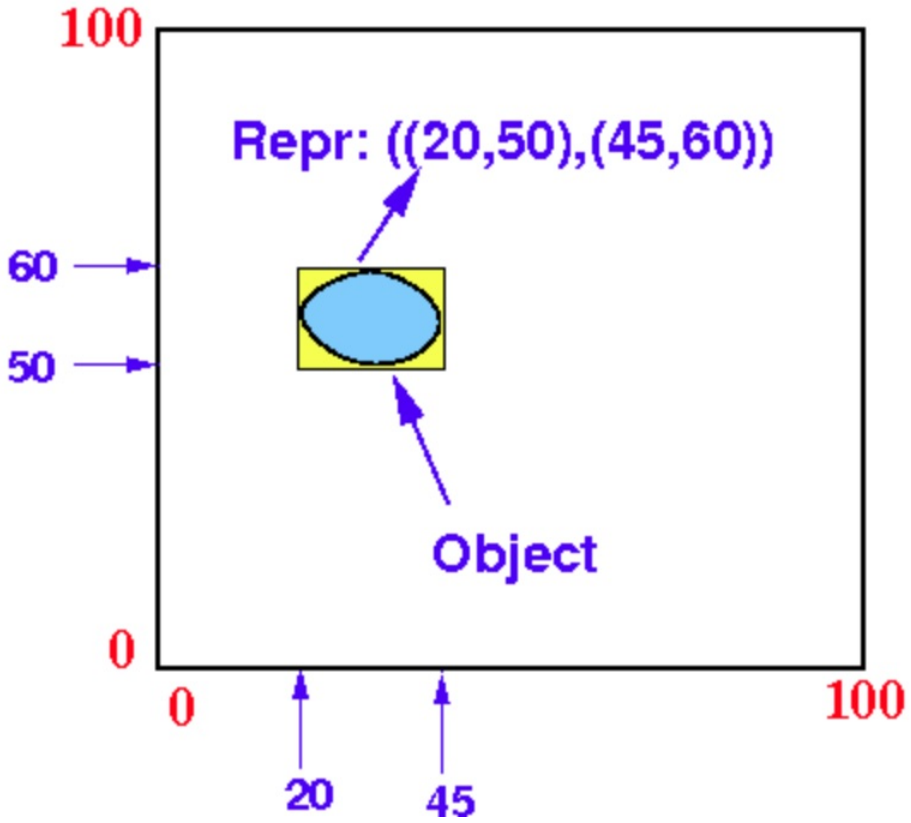
- Execution of the search algorithm:
  - We arrived at a `leaf node`, so we search the `MBBs` in the `leaf node`



  - We find that the `pipeline object` contains the `point`

Lookup algorithm for an object (rectangle) in `R-tree`

- Simplification: We represent an object by its `Minimum Bounding Box`
- Example

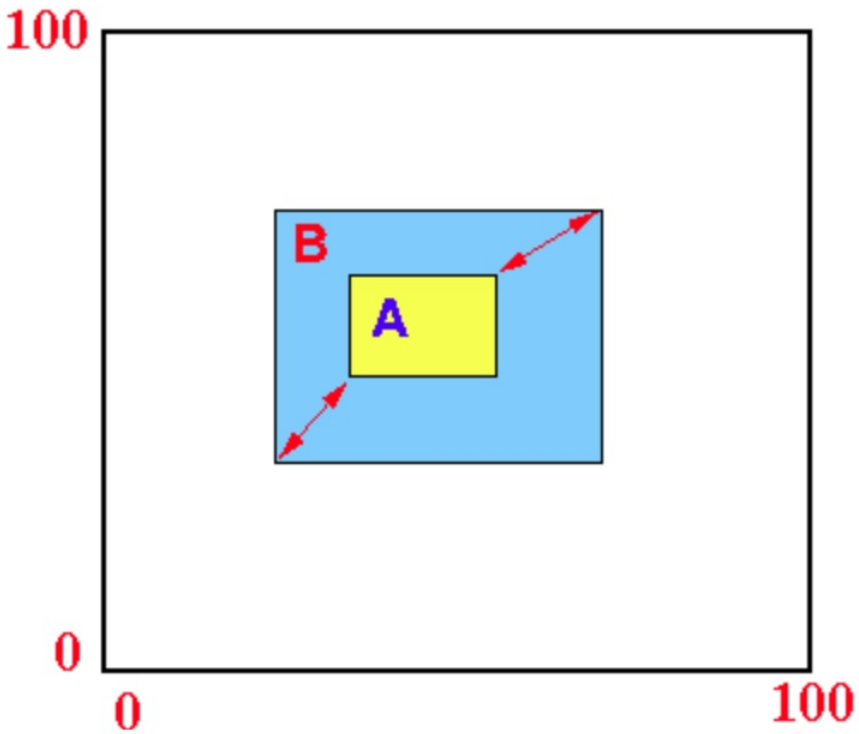


Lookup algorithm for an object (rectangle) in `R-tree`

- Containment relationship of `bounding boxes`
  - A `bounding box A` is contained in a `bounding box B` iff

$$x_{LL}(B) \leq x_{LL}(A) \quad and \quad y_{LL}(B) \leq y_{LL}(A)$$
$$x_{UR}(A) \leq x_{UR}(B) \quad and \quad y_{UR}(A) \leq y_{UR}(B)$$

  - Graphically:



Search algorithm for objects in an `R-tree`

- Is similar to the search algorithm for points using the `containment relationship` for `bounding boxes`
- Algorithm to find `rectangle object` is called using
  - `Lookup(Obj, n, result)`: Look for `object` that contains `rectangle Obj`
    - `Obj`: representation of a rectangle object
    - `n`: current node of search
    - `result`: output

Search algorithm for objects in an R-tree

Algorithm 3 `Lookup(Obj, n, result)`

```
1:  // n = current node of the search in the R-tree
2:  if (n = internal node) then
3:     for each entry (BB, childptr) in internal node n) do
4:        // Look in subtree if Obj is inside bounding box
5:        if Obj ⊆ BB then
6:           Lookup(Obj, childptr, result)
7:        end if
8:     end for
9:  else
10:    //n is a leaf node
11:    for (each object Ob in node n) do
12:       if MBB(Obj) = MBB(Ob) then
13:          Add Ob to result // Object Ob contains Obj
14:          return // Obj found, done
15:       end if
16:    end for
17: end if
```

Insert into the R-tree

- The insert algorithm in the R-tree is very similar to the insert algorithm in the B^+-tree
  - The search key + record pointer index entry is always inserted into a leaf node
- Insert algorithm (to insert an object into a leaf node of the R-tree)
- The insert algorithm is called using
  - Insert((MBB, ObjID), root)
    - MBB: Min. Bounding Box of object
    - ObjID: Object ID (e.g.: record pointer)
    - root: root node of the R-tree

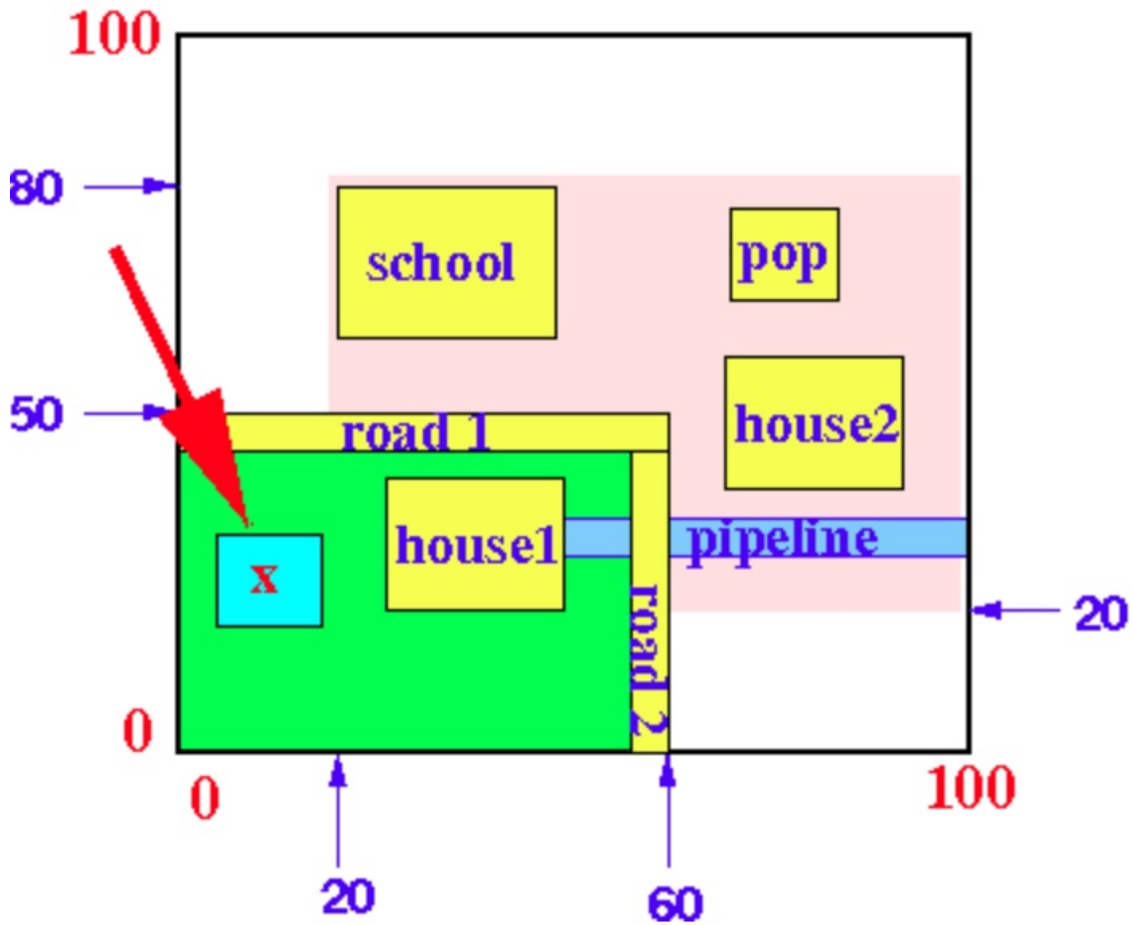
Insert Algorithm

Algorithm 4 `Insert((mbb,ObjID), n)`

```
1:  if (n = internal node ) then
2:      for each entry (BB, childptr) in node n) do
3:          //Insert the object in subtree of the first BB that contains the mbb of the inserted object
4:          if mbb of inserted object ⊆ BB then
5:              // Insert object in this subtree
6:              Insert((mbb,ObjID),childptr) // The recursion WILL have inserted the object
7:              return // Terminates the for-loop
8:          end if
9:      end for
10:     //We ONLY reach here when EVERY BB in n not contain object. Enlarge one of BB so it can store object
11:     Find a BB in n s.t. enlarging BB to contain the new object will add the least amount of area to BB
12:     Update this BB in n to the Enlarged BB
13:     // Now the enlarged BB contains the object
14:     Insert((mbb,ObjID), childptr) // Insert in subtree
15:     return // Done
16: else if (leaf node has space to hold object) then
17:     // We found the leaf node to hold the object
18:     insert(mbb,ObjID) in the leaf node
19: else
20:     Split the objects in the leaf node into set1 and set2
21:     Find the bounding box BB1 for set1 of the objects and bounding box BB2 for set2 of the objects
22:     Replace the parent's (BB,ptr) by (BB1,set1) and (BB2,set2)
23:     //This step can cause the parent node to overflow. Need to split the parent node into 2 internal nodes.
24:     //Use Insert procedure into an internal node
25: end if
```

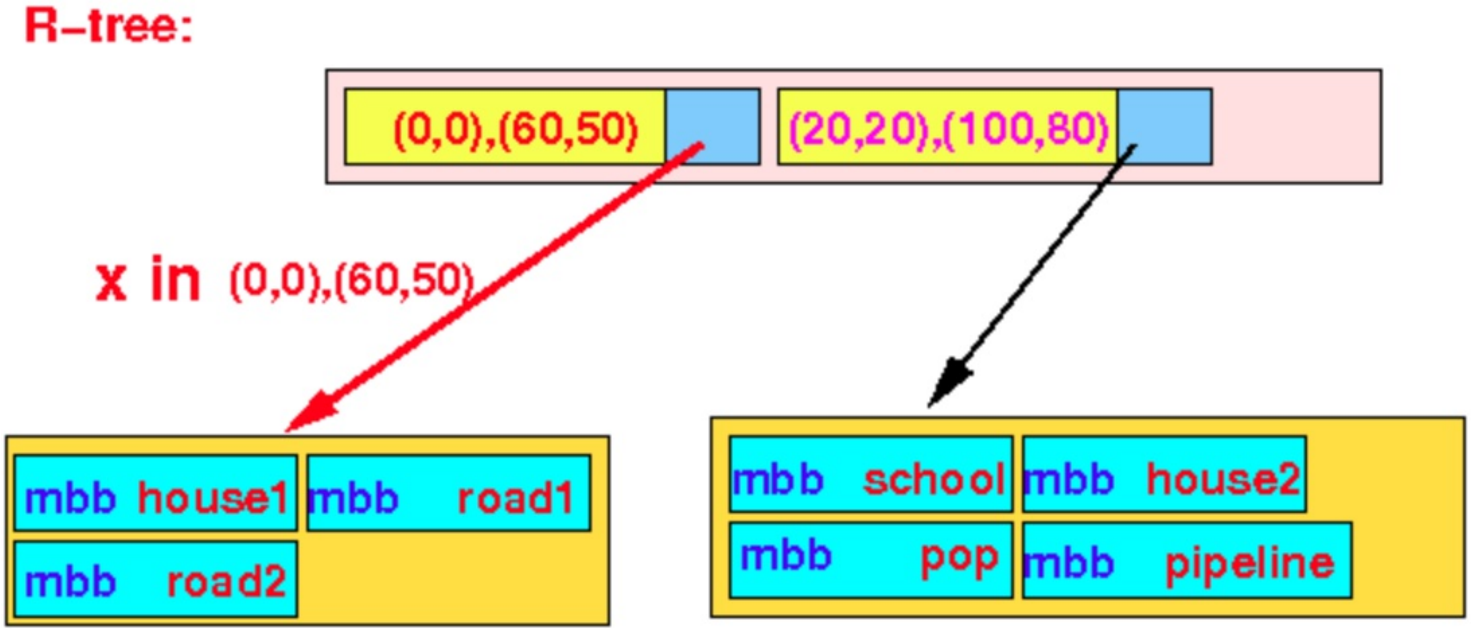
Insert Ex 1: easy case (leaf node has space for object)

- Insert the `object x` into the `R-tree`



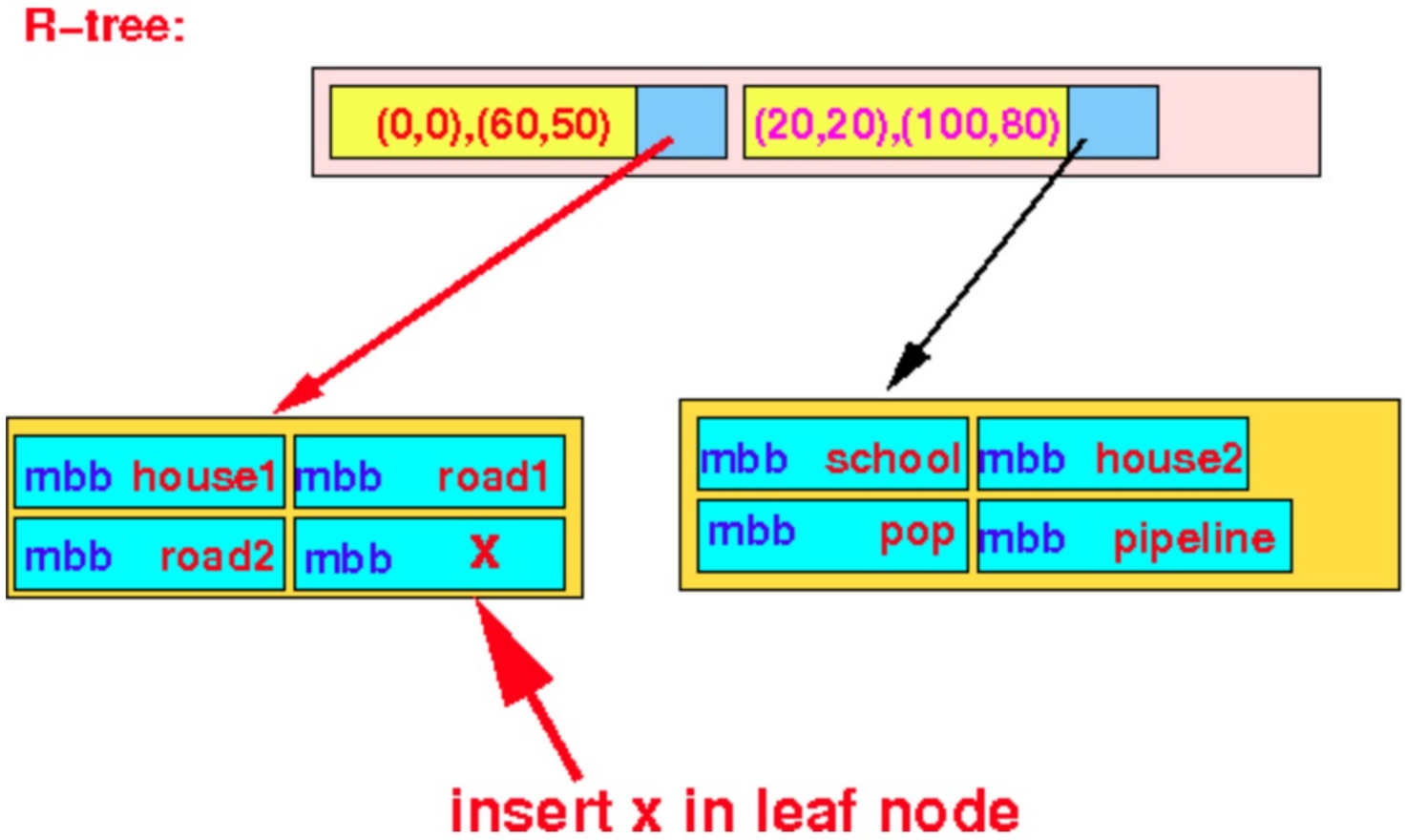
Insert example 1: Steps of the Insert Algorithm

- The first `BB` contains `x`



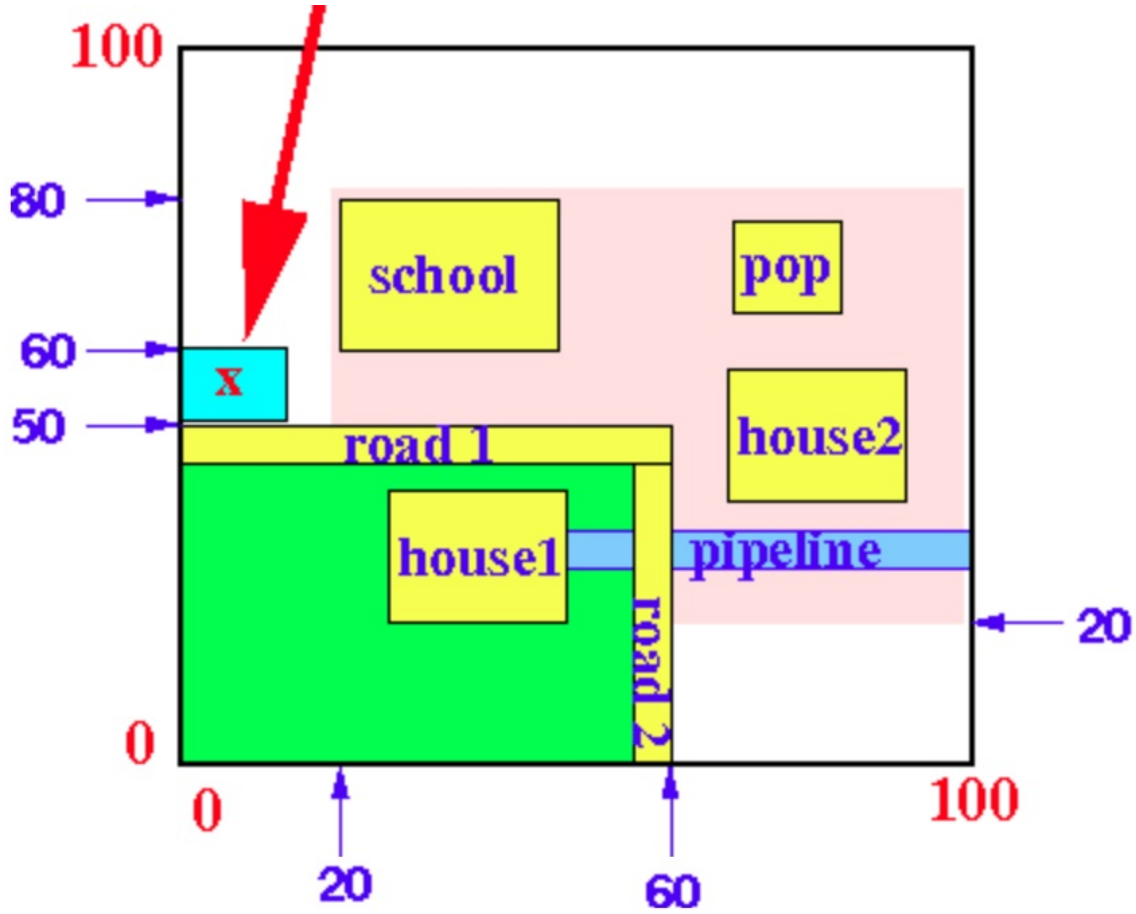
Insert example 1: Steps of the Insert Algorithm

- Because there is space in the `leaf node`, we insert the `object x` there



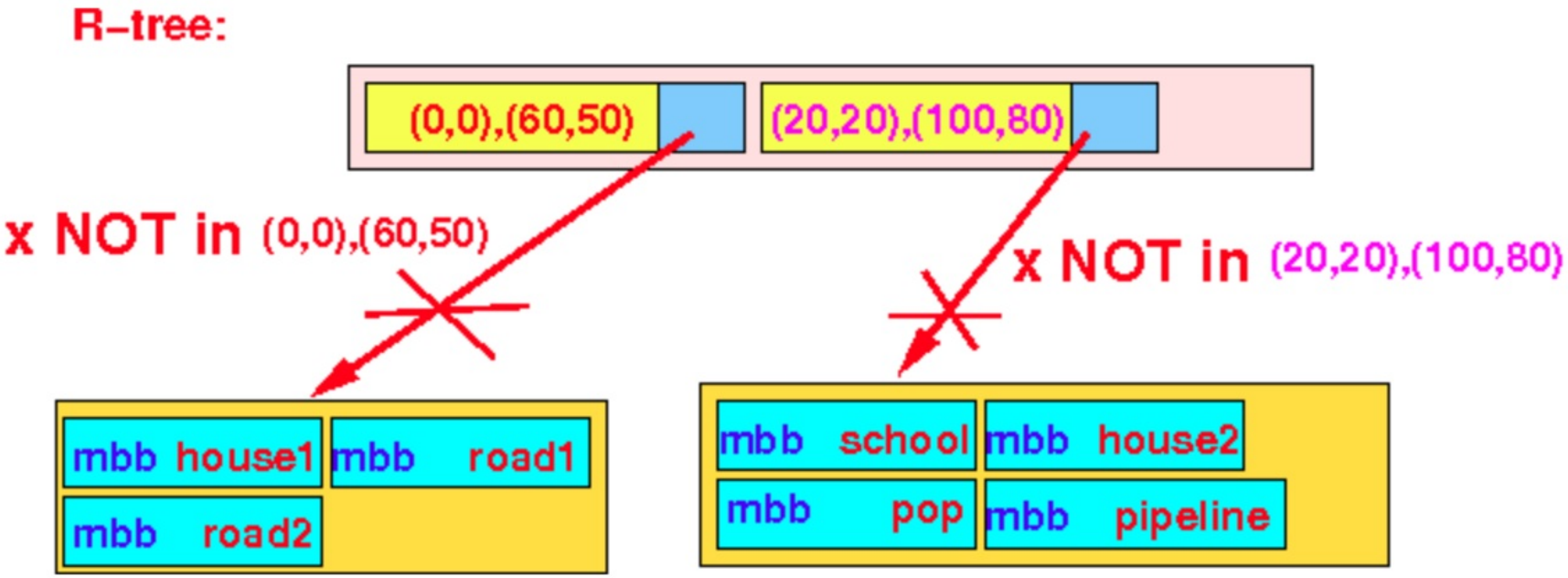
Insert example 2: (augmenting the bounding box)

- Insert the `object x` into the `R-tree`



Insert example 2: Steps of the Insert Algorithm

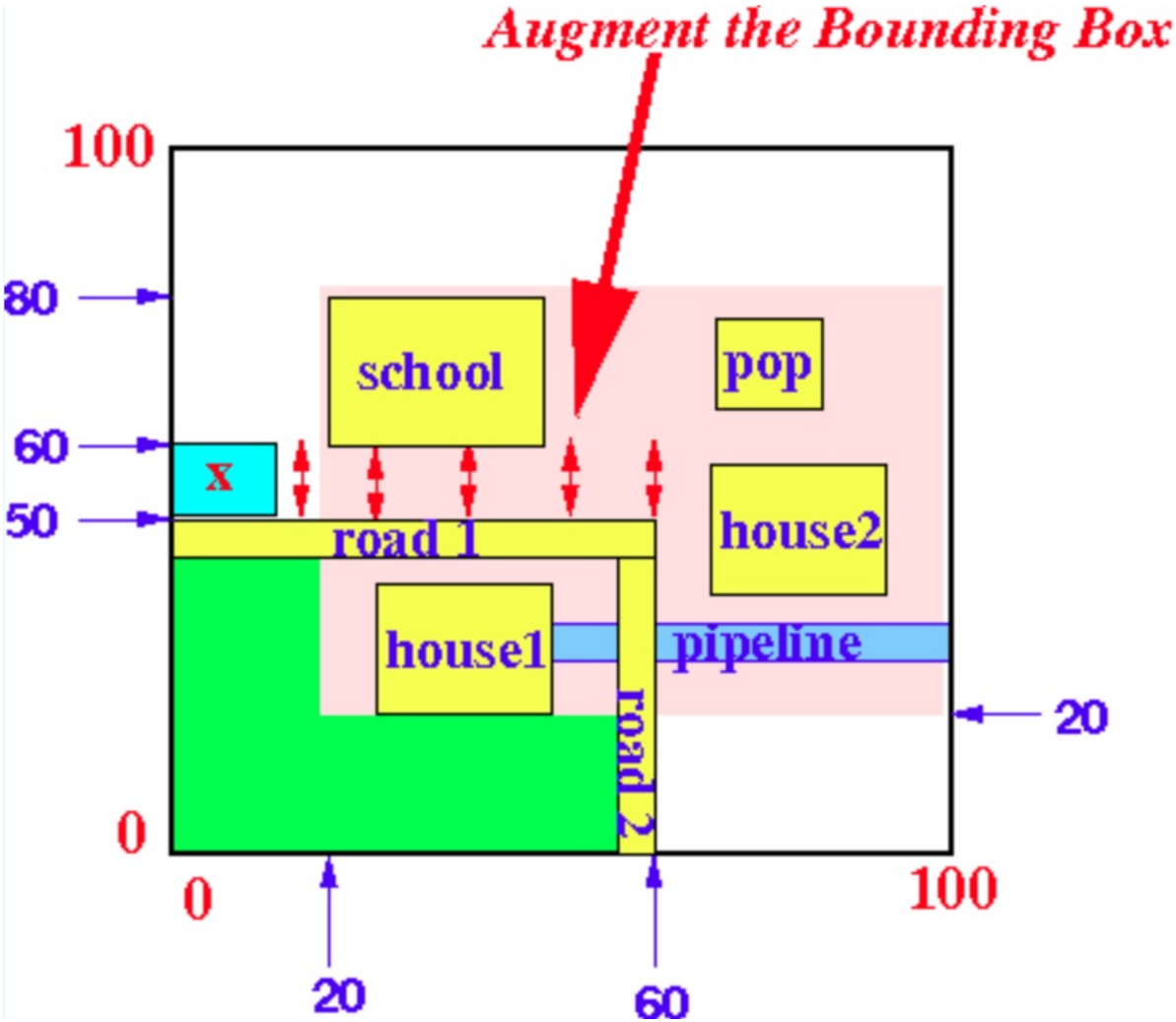
- Find a `bounding box` that contains `x`



- We cannot find any `BB` that contains the `object x`, therefore `Augment one of the bounding boxes`

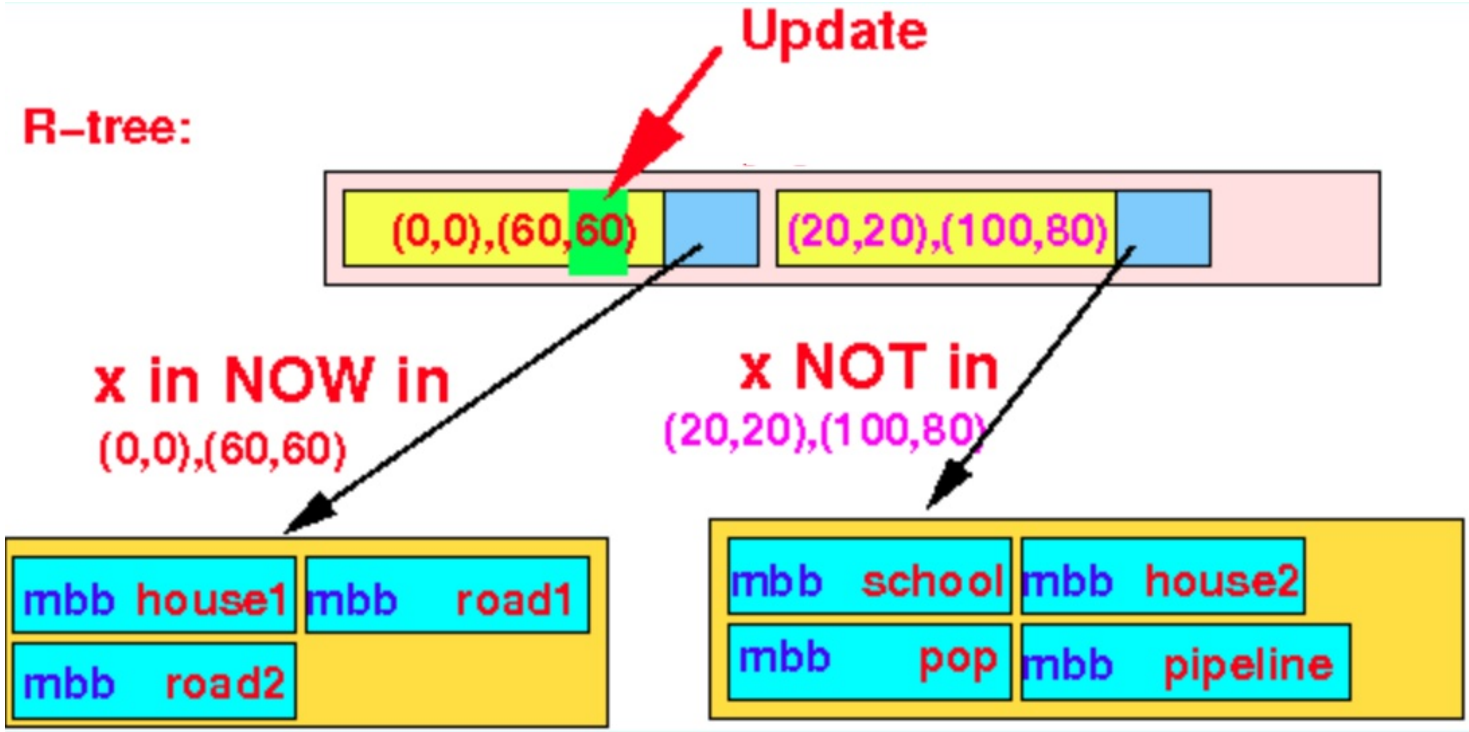
Insert example 2: Steps of the Insert Algorithm

- We can augment the first `BB` to (`(0,0),(60,60)`)



Insert example 2: Steps of the Insert Algorithm

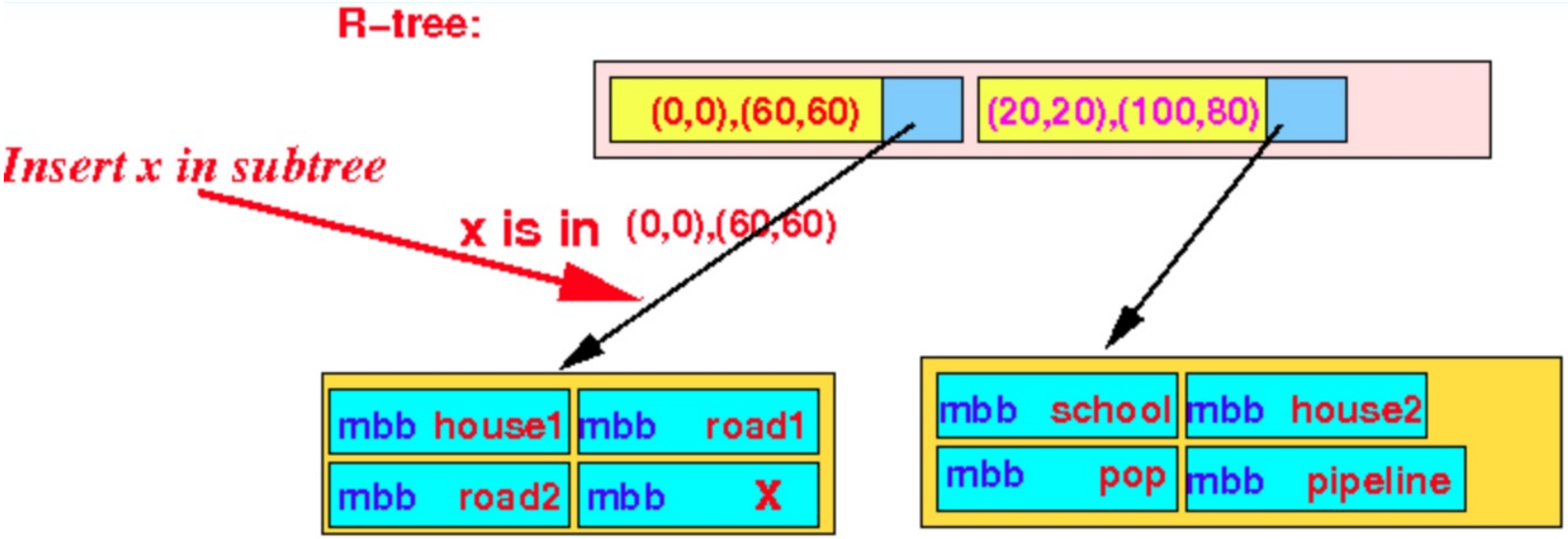
- so that the new will include the `object x`



There is no need to update the `BB` in the `parent node`

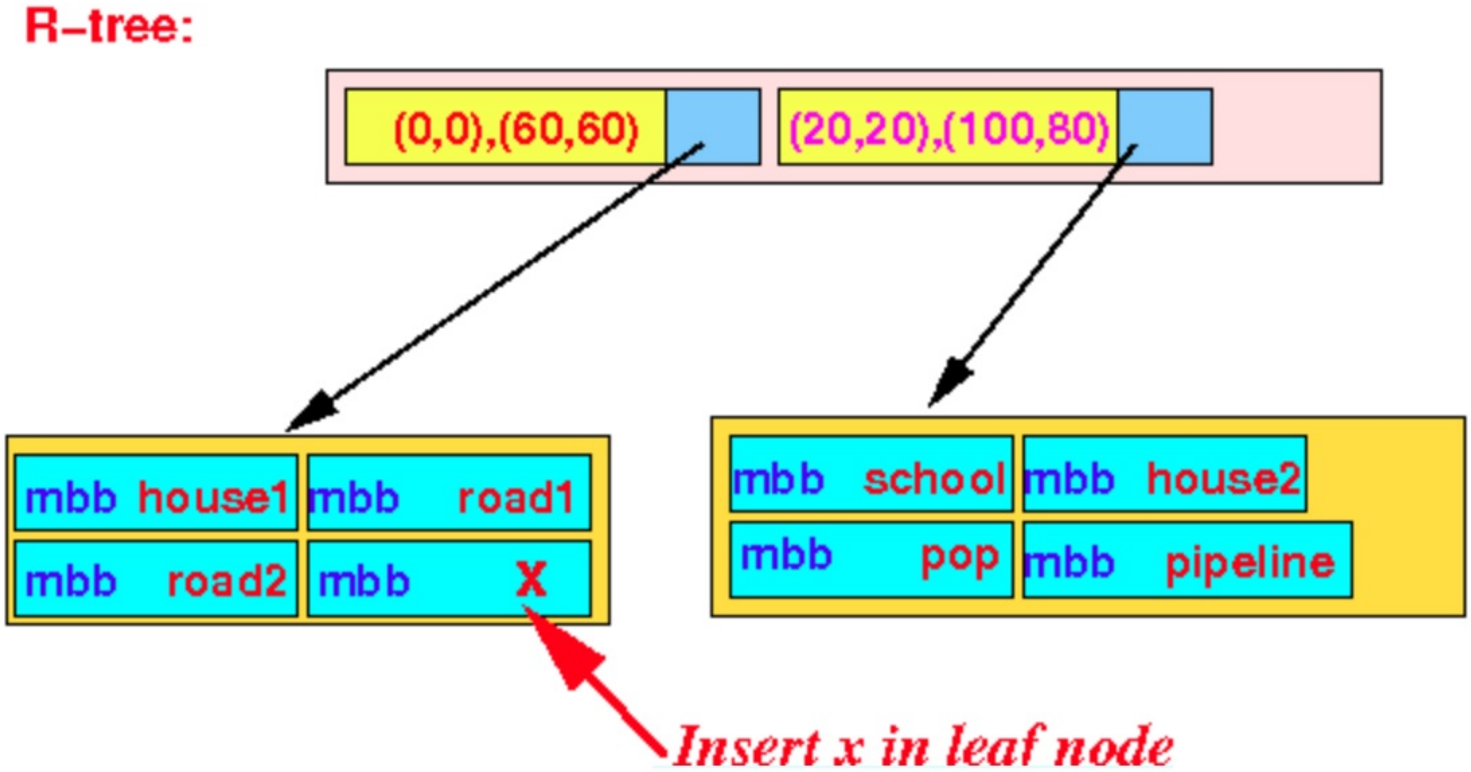
Insert example 2: Steps of the Insert Algorithm

- After augmenting, we can insert the `object x` in the subtree



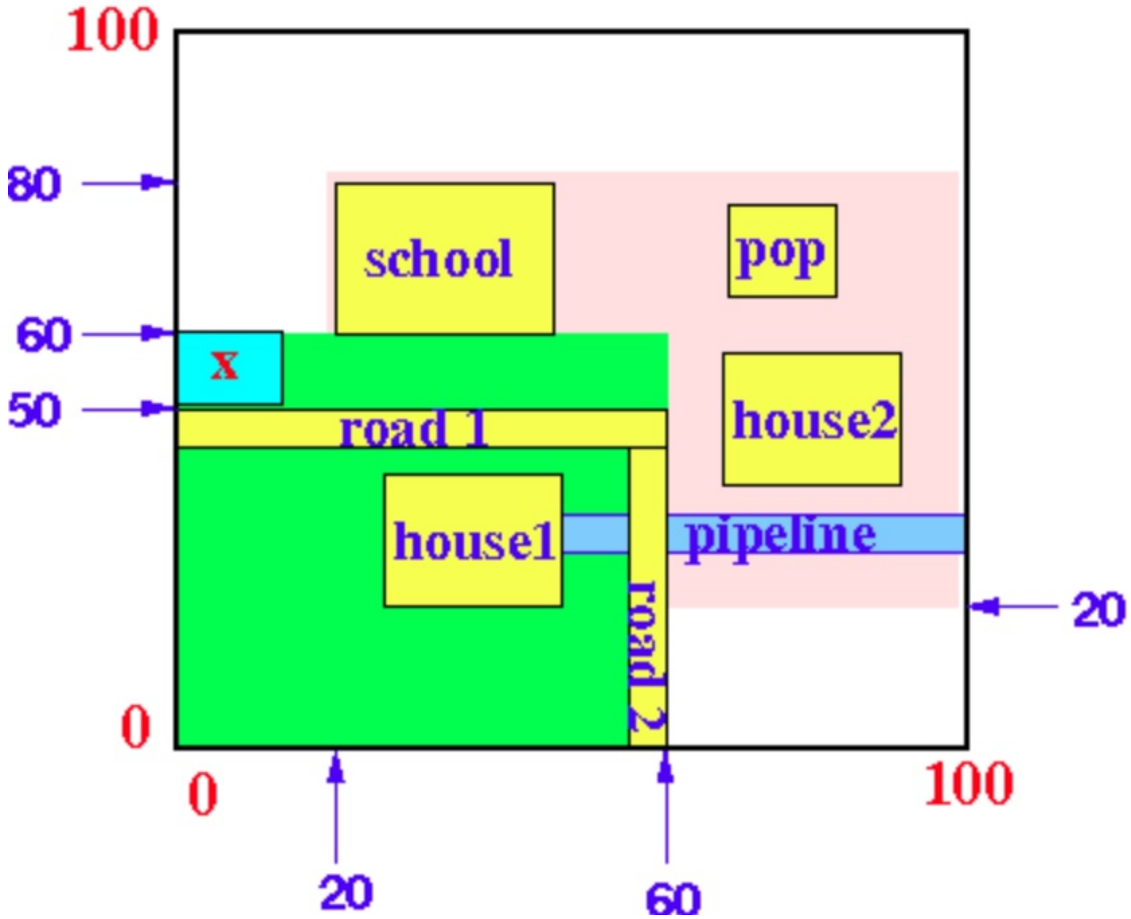
Insert example 2: Steps of the Insert Algorithm

- Which will insert `x` in this `leaf node`



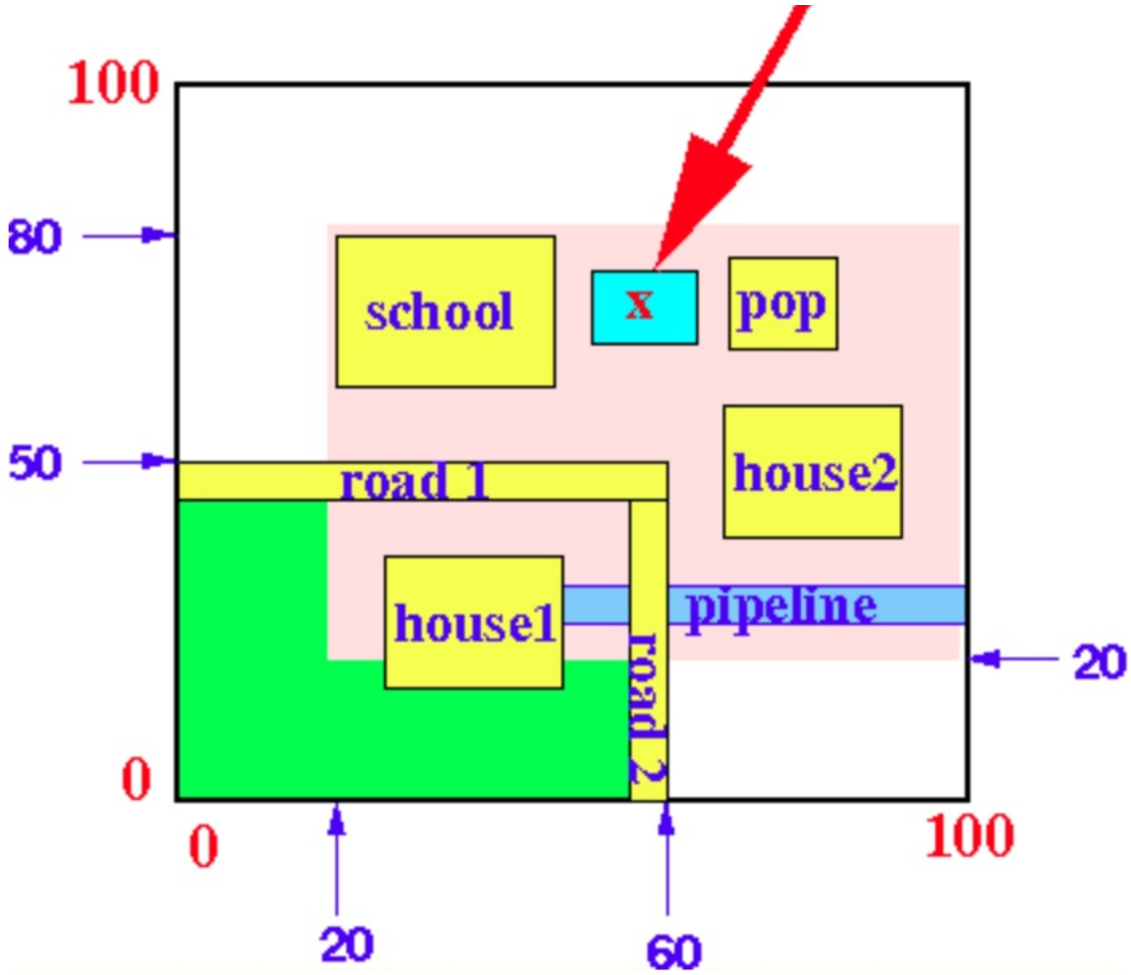
Insert example 2: Steps of the Insert Algorithm

- Which represents the following result



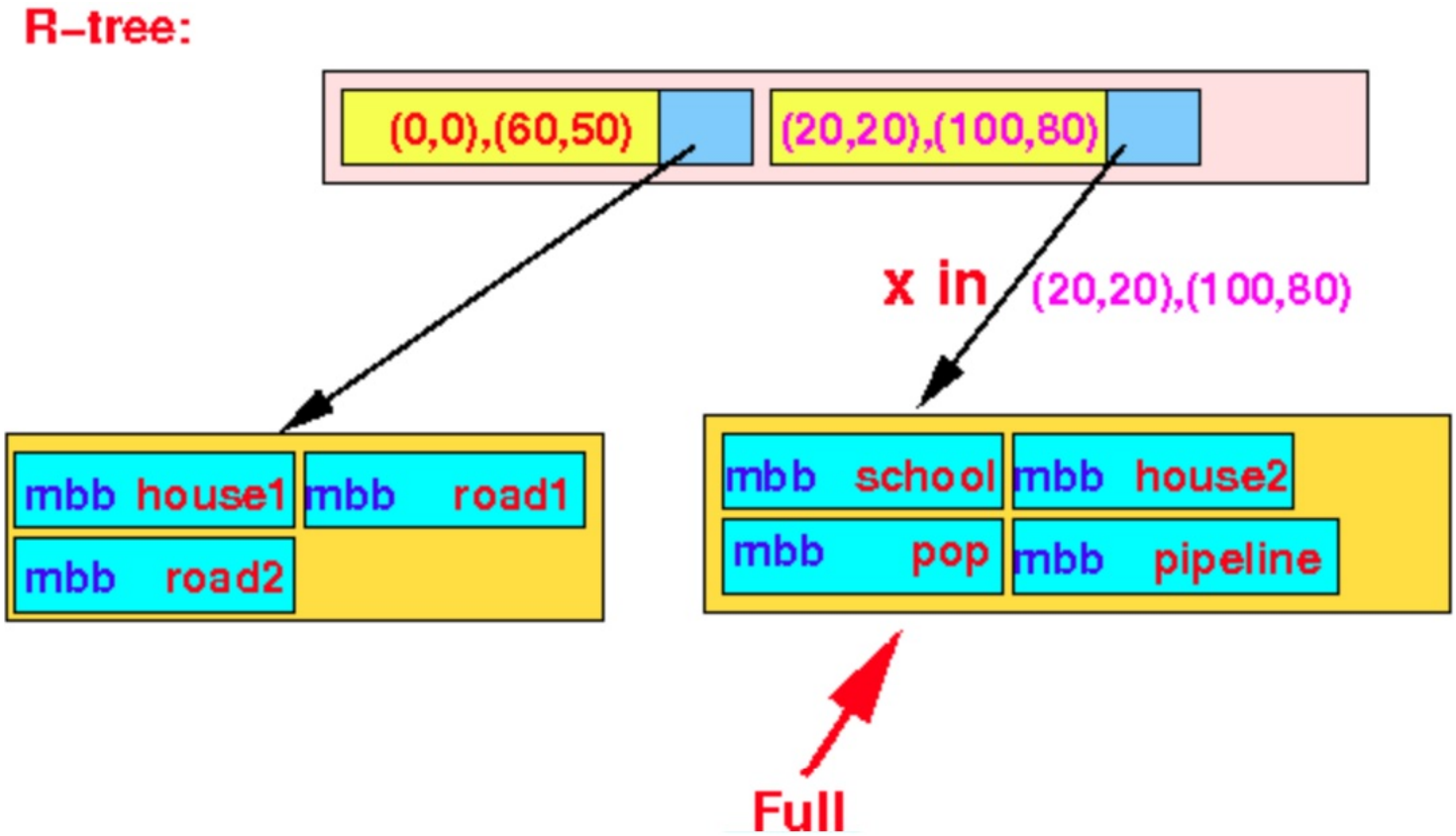
Insert example 3: (split a leaf node)

- Insert the `object x` into the `R-tree`



Insert example 3: Steps of the Insert Algorithm

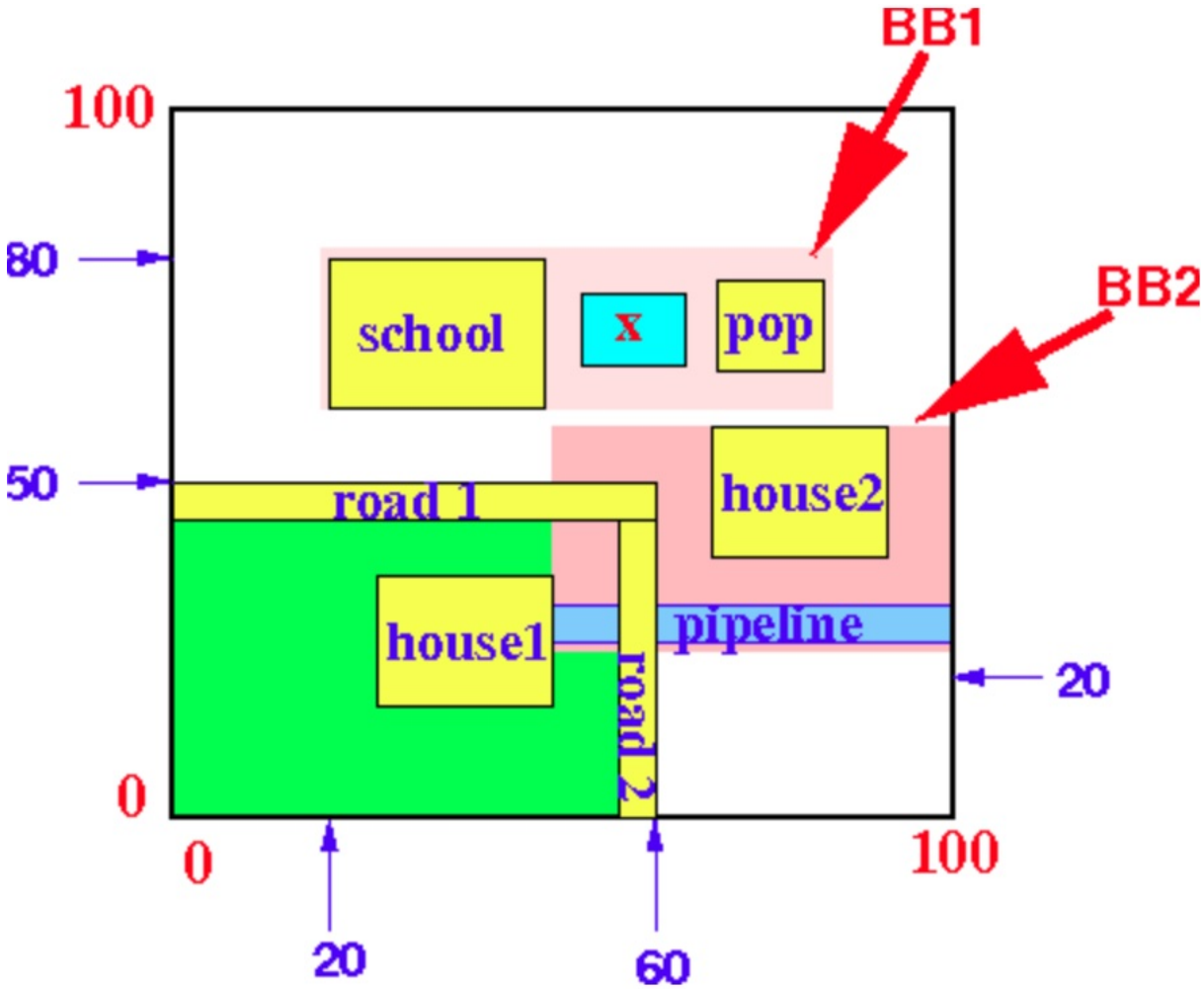
- We find a `BB` that contains the `object x`



- However, the `leaf node` is full. We partition the objects into `2` groups

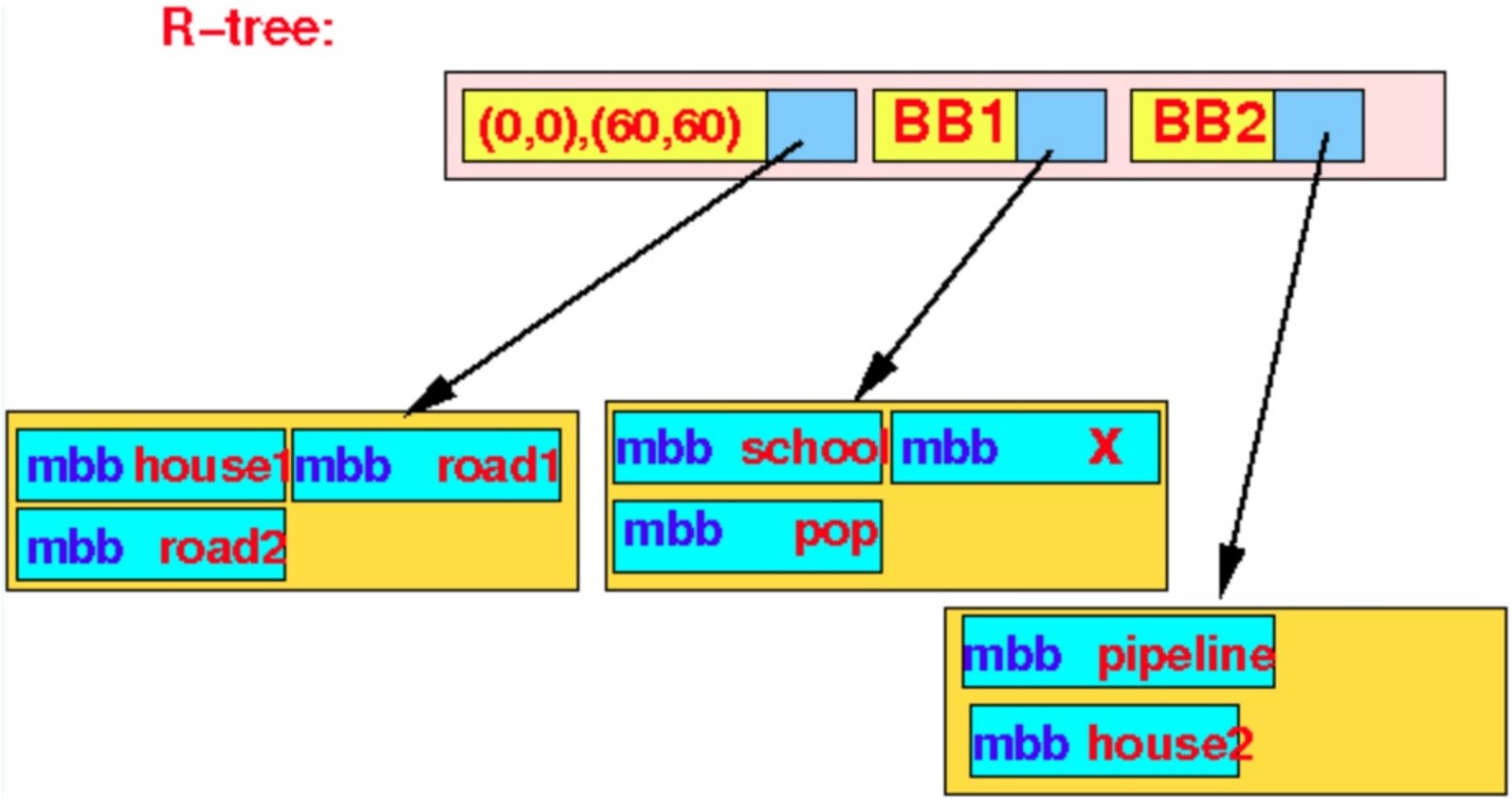
Insert example 3: Steps of the Insert Algorithm

- Suppose we partition the objects as follows



Insert example 3: Steps of the Insert Algorithm

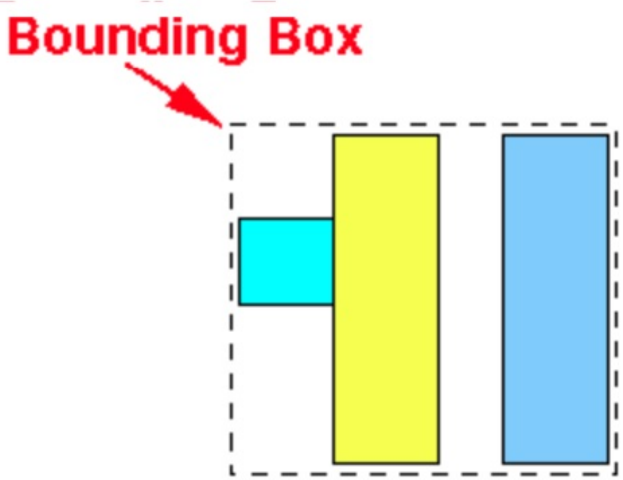
- The new `R-tree` is then



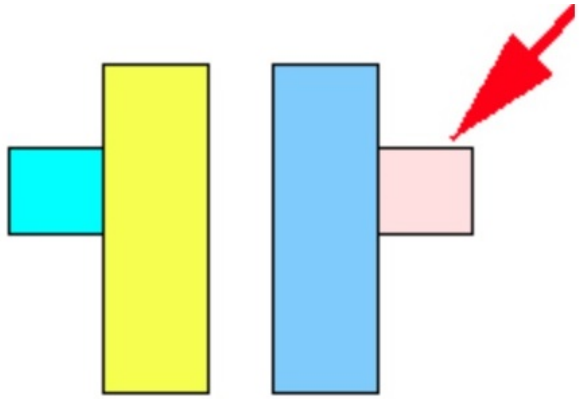
- Notice that: The `bounding box` `((20,20),100,80)` in the `parent node` is replaced by `2` bounding boxes `BB1` and `BB2`

How to partition objects in internal/leaf node

- How to partition (`BB` or `MBB`) objects in an overflow node
  - Suppose a node contains the following (`BB` or `MBB`) objects

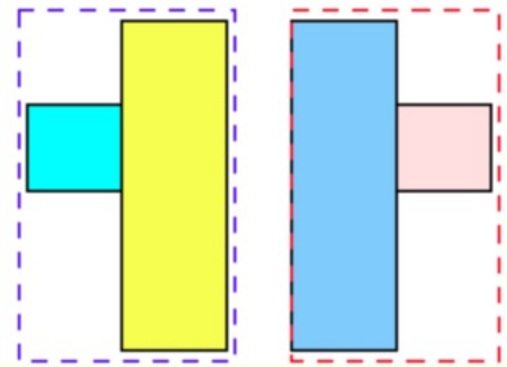


  - We insert the following object

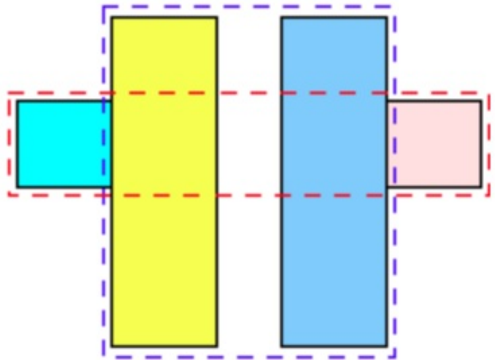


A note on how to partition objects in internal/leaf node

- The node is now overflow and we want to split the objects into 2 groups.
- How should the objects be split
  - An obvious choice is:



  - A not so obvious choice is:



  - This is a superior split because `the total area of the bounding boxes is smaller`

A note on how to partition objects in internal/leaf node

- The goal of the partitioning
  - Fact: The smaller `Bounding Box`, the more precisely the `Bounding Box` describes a `search area`
  - Therefore: When we split the `(search) Bounding Boxes` in a node, we want to `minimize the total area of the bounding boxes of the split`

A naïve object partitioning algorithm

- Problem description
  - Given a set of `n objects`
  - Find the partitioning of the `n objects` into `2 groups` **A** and **B** such that:
    - **AreaOfMBB(objects in A) + AreaOfMBB(objects in B) = minimal**
- Naïve Algorithm
  - `Brute force`: test every possible split
  - Complexity of the `naïve re-distribute algorithm` = $O(2^{n-1})$
    - # subsets of a set of `n` elements = 2^n
    - We need to consider half of all subsets due to symmetry
- There exists a $O(n^2)$ algorithm to find the reasonably good partitioning and is described in R-trees: a dynamic index structure for spatial searching)

Reading

- Spatial Data Types
- Spatial Developer's Guide

Topics

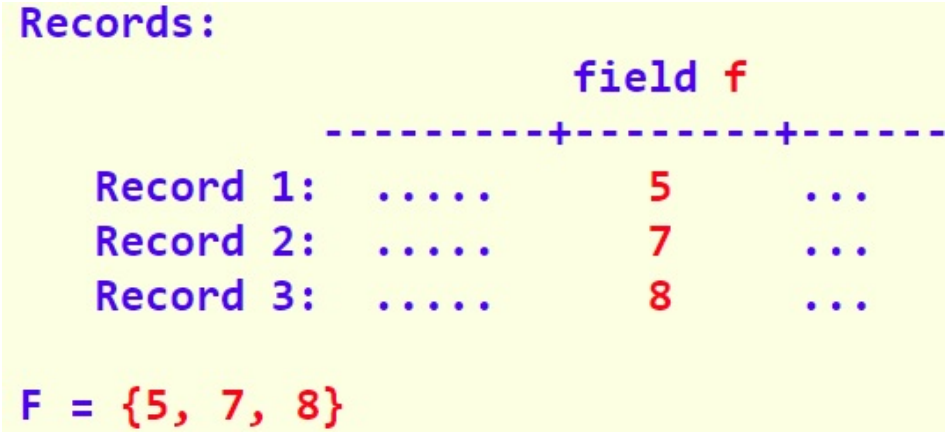
- Additional Index Usage
- `Tries/Radix Trees`
- `Inverted Indexes`
- Multi-dimensional index structures
- `Bitmap Indexes`

Bitmap indexes

- Bitmap indexes are a specialized form of indexing designed for efficient querying involving multiple attributes.
- In a bitmap index:
  - Records in a relation are typically assumed to be sequentially numbered, often starting from 0.
  - Each record is uniquely identified by a position ID, making it easy to retrieve a specific record using its ID.
- Applicable on attributes with a relatively small number of distinct values
  - Gender, country, state, ...
  - Income-level (where income is categorized into a few discrete levels) such as 0-9999, 10000-19999, 20000-50000, 50000- infinity())
- A bitmap is essentially an array of bits, with each bit representing the presence or absence of a specific attribute value for a particular record.

Bitmap indexes

- Current value set (F): refers to the present collection of values that are stored in a specific field, denoted as f, within the records of a dataset or database.
- This set represents the most up-to-date or current values associated with that particular field across all the records in the dataset.
- Example



Bitmap indexes

- Store a separate Bitmap for each unique value for a particular attribute where an offset in the vector corresponds to a tuple.
- Bitmap index of a field f involves creating a collection of bit vectors, one for each unique value v that appears in that field.
- These bit vectors are associated with each unique value and have a length of n, where n corresponds to the number of records in the dataset.
- Each bit vector represents the presence or absence of a specific value v in the field f for each record in the dataset.
- The i^{th} position in a given bit vector corresponds to the i^{th} tuple (record) in the dataset.

Bitmap indexes

- For a specific value v, the bit vector is constructed such that each x_i (where i ranges from 1 to n) is set to 1 if the ith record's field f is equal to v, and it is set to 0 if it is not equal to v:

$$x_i = \begin{cases} 1 & \text{if the } i\text{th record's field } f \text{ is equal to } v \\ 0 & \text{if it is not equal to } v \end{cases}$$

- To optimize memory usage and allocation, these bit vectors are typically segmented into smaller chunks or blocks, preventing the need for large contiguous memory allocations.

Bitmap indexes: Example

- A file has 6 records

Fields:	A	B
record 1:	30	foo
record 2:	30	bar
record 3:	40	baz
record 4:	50	foo
record 5:	40	bar
record 6:	30	baz

- The `bitmap index` for the `field A` is

value	123456	
30	110001	<---- bit vector
40	001010	
50	000100	

Explanation:

The value 30 appears in the records: 1, 2, 6
So: bit #1, #2 and #6 are set

Bitmap indexes: Example

- A file has 6 records

```
  Fields:          A          B
-------------------------------
  record 1:       30        foo
  record 2:       30        bar
  record 3:       40        baz
  record 4:       50        foo
  record 5:       40        bar
  record 6:       30        baz
```

- The `bitmap index` for the `field B` is

```
    value          123456
  ----------------------------
      foo          100100          <---- bit vector
      bar          010010
      baz          001001

Explanation:

    The value foo appears in the records: 1, 4
    So: bit   #1 and #4 are set
```

Bitmap indexes: Example: people who buy jewelry

- Data on people who buy jewelry

(age, salary (in $1,000))

1(25,60)	2(45,60)	3(50,75)	4(50,100)
5(50,120)	6(70,110)	7(85,140)	8(30,260)
9(25,400)	10(45,350)	11(50,275)	12(60,260)

- The `bitmap index` on `age` is

Value	123456789012
25	100000001000
30	000000010000
45	010000000100
50	001110000010
60	000000000001
70	000001000000
85	000000100000

Bitmap indexes: Example: people who buy jewelry

- Data on people who buy jewelry

(age, salary (in $1,000))

1(25,60)	2(45,60)	3(50,75)	4(50,100)
5(50,120)	6(70,110)	7(85,140)	8(30,260)
9(25,400)	10(45,350)	11(50,275)	12(60,260)

- The `bitmap index` on `salary` is

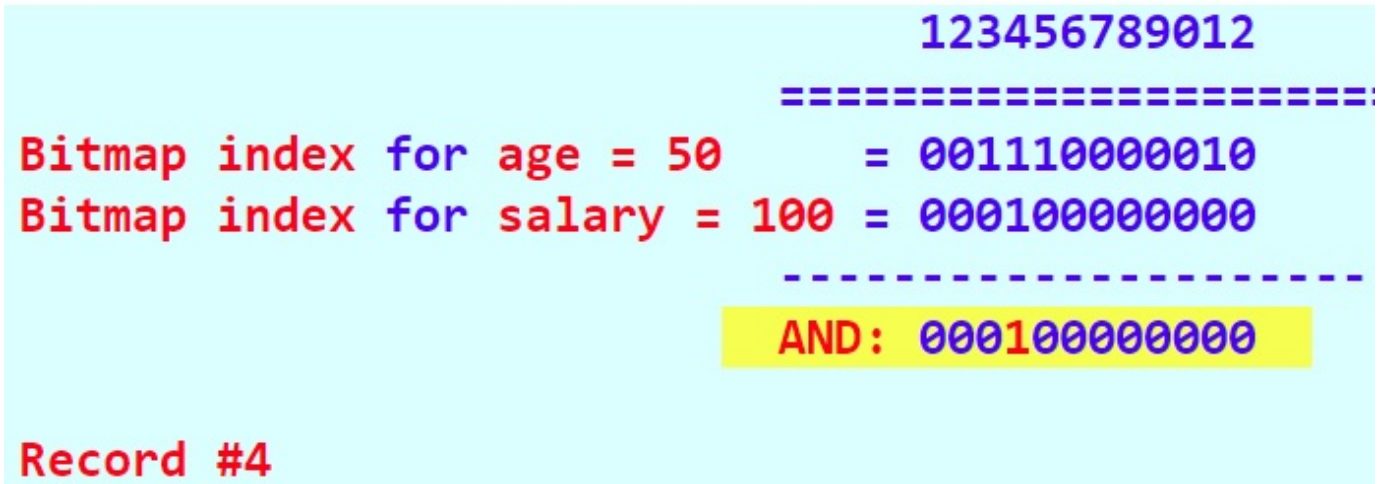
Value	123456789012
60	110000000000
75	001000000000
100	000100000000
110	000001000000
120	000010000000
140	000000100000
260	000000010001
275	000000000010
350	000000000100
400	000000001000

Using Bitmap indexes

- Example query:

  `Find people (who buy jewelry) such that age=50 and salary=100`

- Answer:

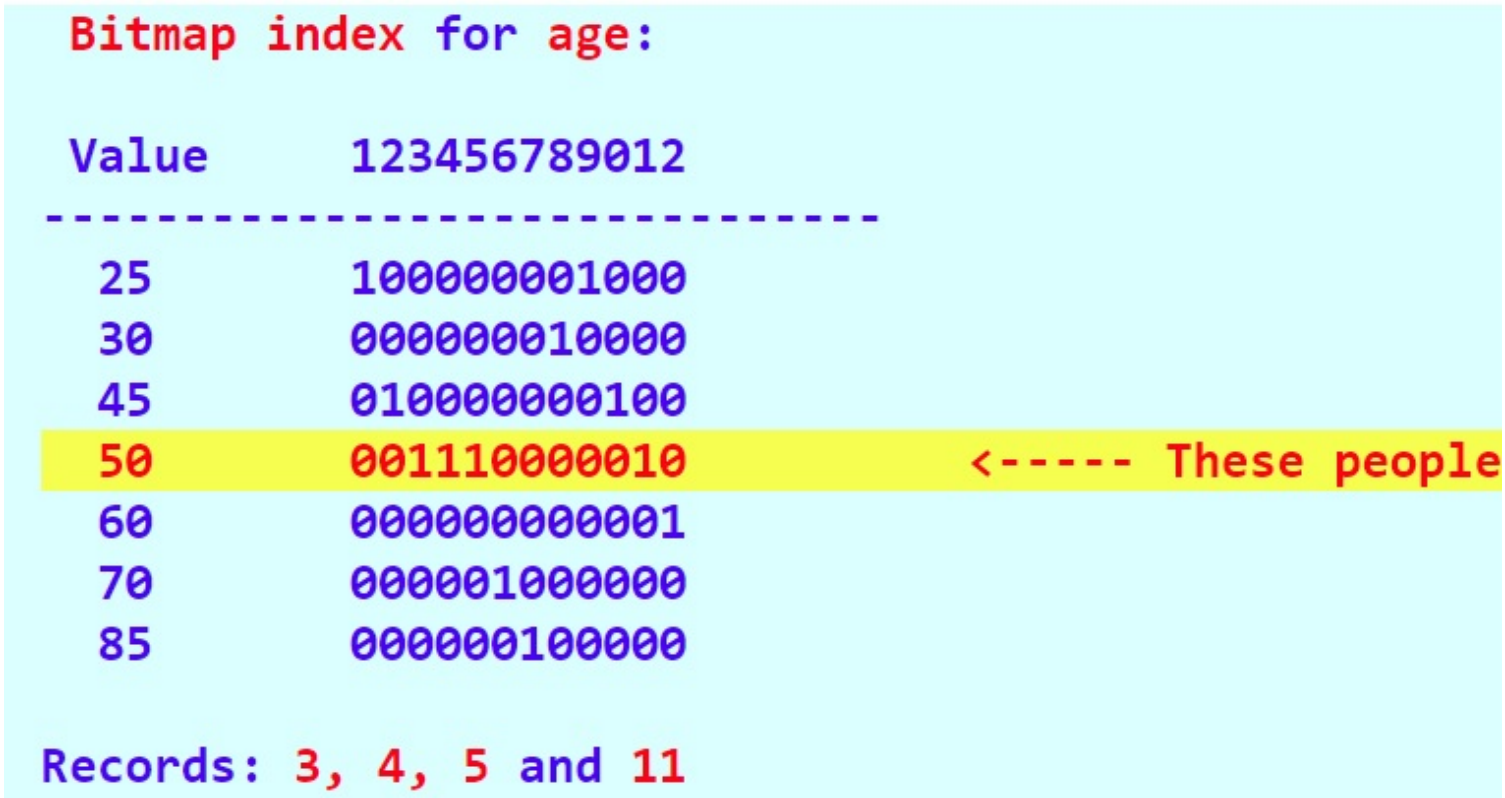


Multi-dimensional nature of `Bitmap indexes`

- When you have `bitmap indexes` on different fields (of a record), the `bitmap indexes` are `multi-dimensional`
- There are some `multi-dimensional` queries that can be answered efficiently using `bitmap indexes`

1) Partial Match queries using `Bitmap indexes`

- Query: `Find people (buyers of jewelry) whose age = 50`
- Solution:



2) Range Match queries using `Bitmap indexes`

- Query: `Find people (buyers of jewelry) where` $45 \leq age \leq 50$, $100 \leq salary \leq 200$
- Solution:

```
 Bitmap index for age:

 Value      123456789012
-------------------------------
   25       100000001000
   30       000000010000
   45       010000000100        45 ≤ age ≤ 50
   50       001110000010
   60       000000000001
   70       000001000000
   85       000000100000

Or value = 011110000110
```

2) Range Match queries using Bitmap indexes

- Query: Find people (buyers of jewelry) where $45 \leq age \leq 50$, $100 \leq salary \leq 200$
- Solution:

Bitmap index for salary:

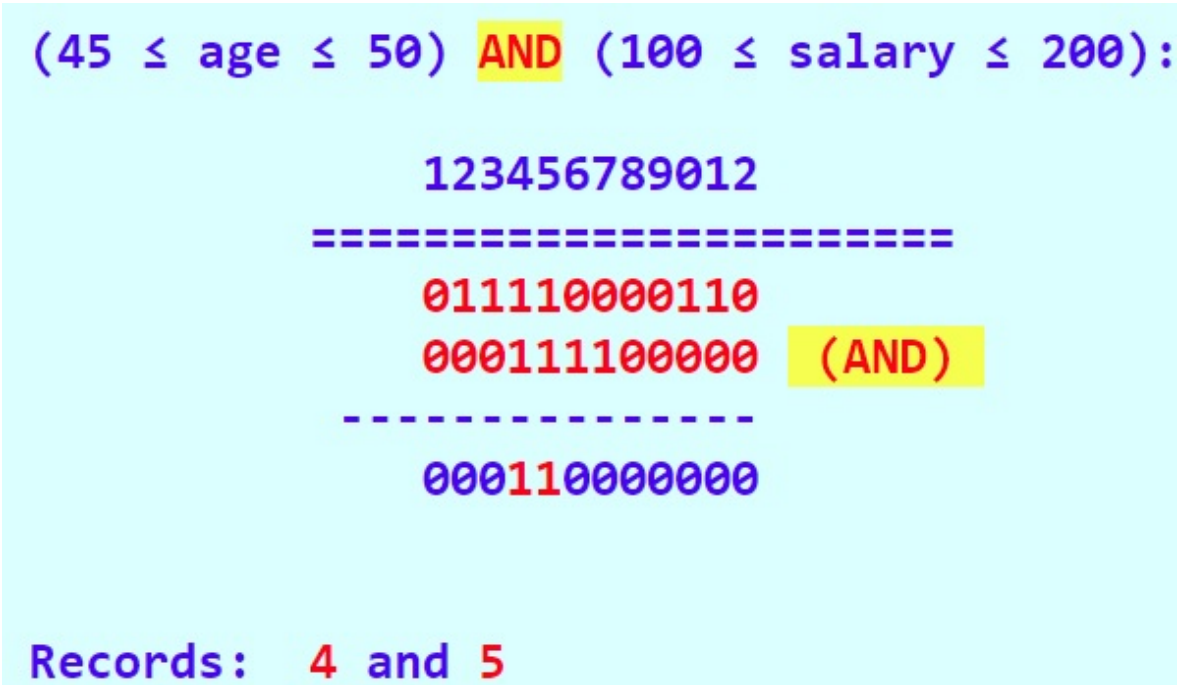
Value	123456789012
60	110000000000
75	001000000000
100	000100000000
110	000001000000
120	000010000000
140	000000100000
260	000000010001
275	000000000010
350	000000000100
400	000000001000

100 ≤ salary ≤ 200

Or value = 000111100000

2) Range Match queries using `Bitmap indexes`

- Query: `Find people (buyers of jewelry) where` $45 \leq age \leq 50$, $100 \leq salary \leq 200$
- Solution:



Bitmap indexes

- Fast for read intensive workloads
  - Used a lot in data warehousing
    - Data warehousing involves the collection and storage of large volumes of data for analytical purposes.
    - These queries often involve filtering and aggregating data based on various criteria, which is where bitmap indexes shine.
- Often build on the fly during `query processing`
  - As we will see later in class

Compression

- Observation
  - Each record has one value in the indexed attribute
  - For a dataset with n records and domain (distinct values) of size $|D|$
    - If D is large, then `1's` in a `bit-vector` will be very rare (only $\frac{1}{|D|}$ bits are 1)
    - $\Rightarrow$ leading to a waste of space
- Solution
  - Compress data. Uncompressed indexes are large
  - Need to make sure that **and** and **or** are still fast

Types of Compression

- Lossy
  - Compression that involves the removal of data.
  - Not used where all data is important.
- Lossless
  - Compression that involves no removal of data, but it's rearranged to be more efficient.

Database Compression[6]

- **Goal #1:** Must produce fixed-length values. Allows us to be efficient while accessing tuples.
- **Goal #2:** Allow the DBMS to postpone decompression as long as possible during query execution. Operate directly on compressed data.
- **Goal #3:** Must be a lossless scheme. No data should be lost during this transformation.

[6] Slide Credit: Joy Arulraj, GATech

Lossless vs. Lossy Compression

- When a DBMS uses compression, it is always lossless because people don't like losing data.
- Any kind of lossy compression is has to be performed at the application level.
  - Example: Sensor data. Readings are taken every second, but we may only store average across every minute.

Main Memory Database Systems

- Reading
  - `3.2 Indexing`: Main Memory Database Systems
  - Integrating Compression and Execution in Column-Oriented Database Systems
  - Dictionary-based Order-preserving String Compression for Main Memory Column Stores
  - A Comparison of Adaptive Radix Trees and Hash Tables