Comp 5311 Database Management Systems

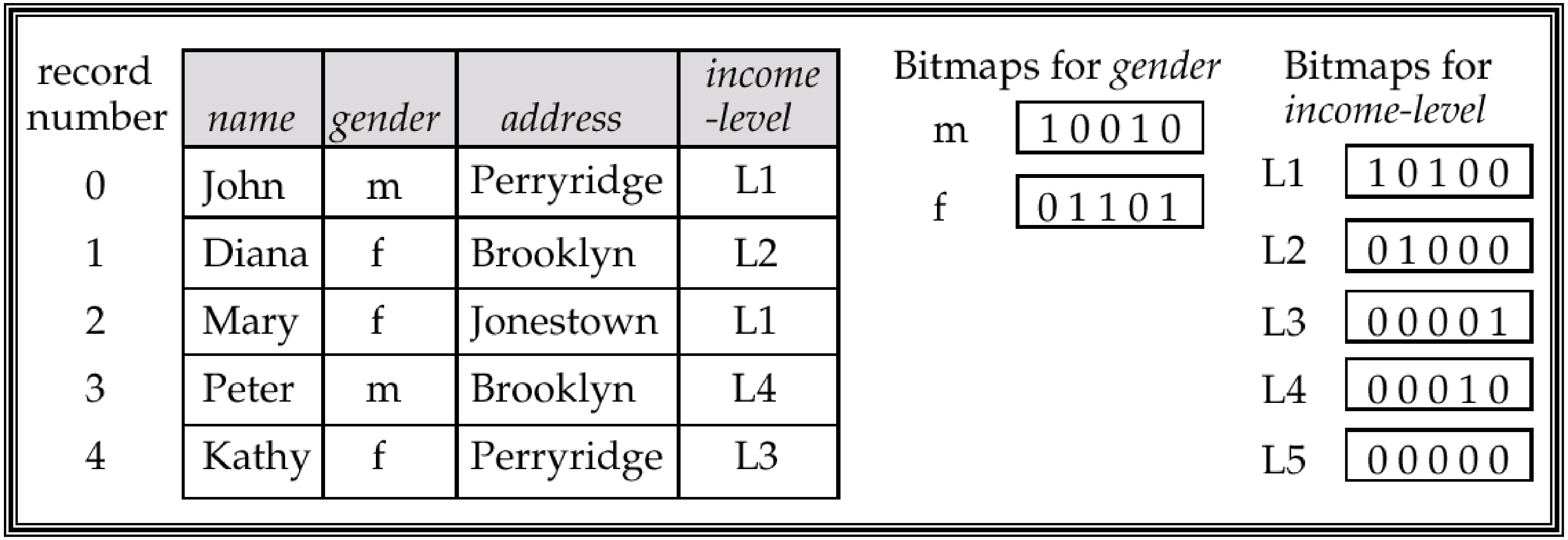
11. Other Indexes, Selection Processing
External Sorting

Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- Records in a relation are assumed to be numbered sequentially from, say, 0
  - Given a number *n* it must be easy to retrieve record *n*
- Applicable on attributes that take on a relatively small number of distinct values
  - E.g. gender, country, state, …
  - E.g. income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000-infinity)
- A bitmap is simply an array of bits

Bitmap Indices (Cont.)

- In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute
  - Bitmap has as many bits as records
  - In a bitmap for value *v*, the bit for a record is 1 if the record has the value *v* for the attribute, and is 0 otherwise



record number	*name*	*gender*	*address*	*income -level*
0	John	m	Perryridge	L1
1	Diana	f	Brooklyn	L2
2	Mary	f	Jonestown	L1
3	Peter	m	Brooklyn	L4
4	Kathy	f	Perryridge	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income-level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Bitmap Indices (Cont.)

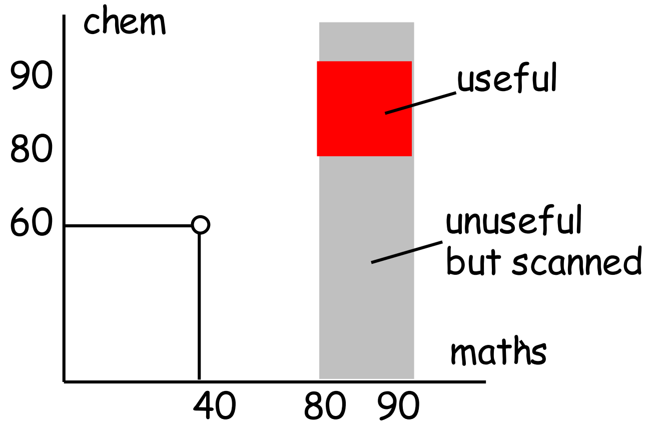
- Bitmap indices are useful for queries on multiple attributes
  - not particularly useful for single attribute queries
- Queries are answered using bitmap operations
  - Intersection (and)
  - Union (or)
  - Complementation (not)
- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
  - Males with income level L1:   10010 AND 10100 = 10000
    - Can then retrieve required tuples.
    - *Counting* number of matching tuples is even faster (especially useful for aggregation queries)

Bitmap Indices (Cont.)

- Bitmap indices generally very small compared with relation size
- Deletion needs to be handled properly
  - Existence bitmap to note if there is a valid record at a record location
  - Needed for complementation
    - not(*A*=*v*): *(NOT bitmap-A-v) AND ExistenceBitmap*
- Should keep bitmaps for all values, even null values
  - To correctly handle SQL null semantics for NOT(*A*=*v*):
    - intersect above result with (NOT *bitmap-A-Null*)

Multi-dimensional Range Search

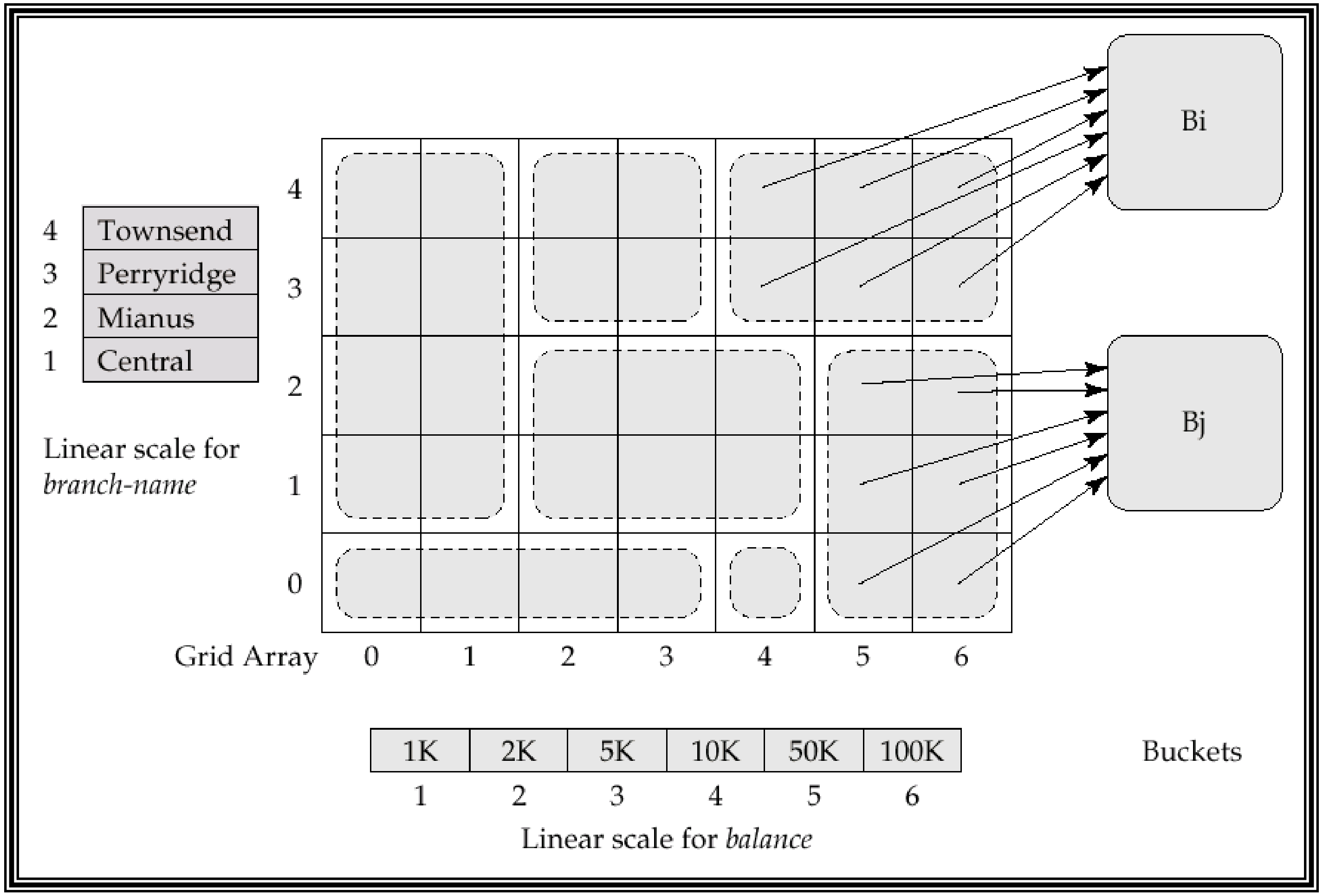
- SELECT stu_id FROM scores WHERE maths in [80,90] & chem in [80,90]
- If we have an index on maths:
  - First retrieve all tuples satisfying maths in [80,90]
  - Check each such tuple about chemistry
- Problem: Many tuples satisfying maths in [80,90] do not satisfy chem in [80,90] but need to be checked anyway
- If we have a B+-tree on <maths, chem> we cannot efficiently process the query WHERE maths in [80,90] & chem = 80 (fetch many records that satisfy the first but not the second condition).
- if we have two indexes on maths and chem, we can do pointer intersection (before retrieving the actual records)



Grid Files

- Structure used to speed the processing of general multiple search-key queries involving one or more comparison operators.
- The grid file has a single grid array and one linear scale for each search-key attribute. The grid array has number of dimensions equal to number of search-key attributes.
- Multiple cells of grid array can point to same bucket
- To find the bucket for a search-key value, locate the row and column of its cell using the linear scales and follow pointer

Example Grid File for *account*



Queries on a Grid File

- A grid file on two attributes *A* and *B* can handle queries of all following forms with reasonable efficiency
  - $(a_1 \le A \le a_2)$
  - $(b_1 \le B \le b_2)$
  - $(a_1 \le A \le a_2 \wedge b_1 \le B \le b_2)$,.
- E.g., to answer $(a_1 \le A \le a_2 \wedge b_1 \le B \le b_2)$, use linear scales to find corresponding candidate grid array cells, and look up all the buckets pointed to from those cells.

Grid Files (Cont.)

- During insertion, if a bucket becomes full, new bucket can be created if more than one cell points to it.
  - Idea similar to extendible hashing, but on multiple dimensions
  - If only one cell points to it, either an overflow bucket must be created or the grid size must be increased
- Linear scales must be chosen to uniformly distribute records across cells.
  - Otherwise there will be too many overflow buckets.
- Periodic re-organization to increase grid size will help.
  - But reorganization can be very expensive.
- Space overhead of grid array can be high.

R-trees

A height balanced tree similar to B$^+$-tree . Currently, the most popular multidimensional index (included in Oracle and several commercial DBMS).

For indexing multiple attributes of relational records or spatial objects.

Objects/records are grouped in nodes (disk pages/blocks) according to spatial proximity.

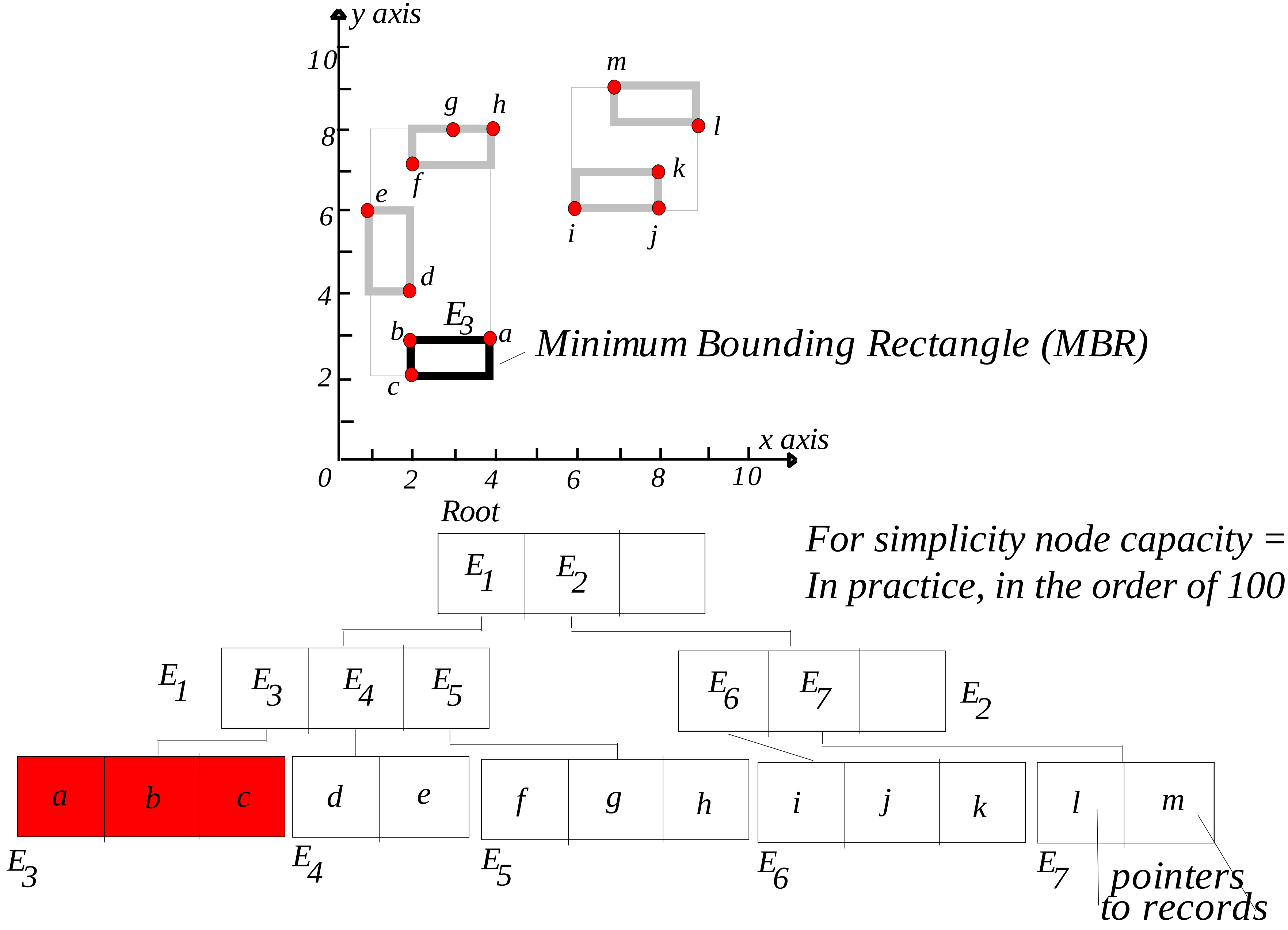
Minimum node utilization *m* is variable, usually 40% of maximum capacity *b*. For typical page sizes (e.g., 4Kbytes or more), *m* and *b are* in the order of hundreds.

Each entry is a pair (*MBR*, *ptr*), that contains:

• a pointer *ptr* to an indexed object or a lower level node

• the *minimum bounding rectangle* (*MBR*) of the pointed object or node

R-Tree



For simplicity node capacity = 3

In practice, in the order of 100

Quadratic R-tree *split* algorithm

1. Apply PickSeeds to choose two entries to be the first elements of the groups. Assign each to a group
2. If all entries have been assigned, stop. If one group has so few entries that all the rest must be assigned to it, assign them and stop
3. Invoke Algorithm PickNext to choose the next entry to assign. Add it to the group whose MBR will have to be enlarged least to accommodate it. Resolve ties by adding the entry to the group with smaller MBR, then to the one with fewer entries, then to either
4. Go to 2

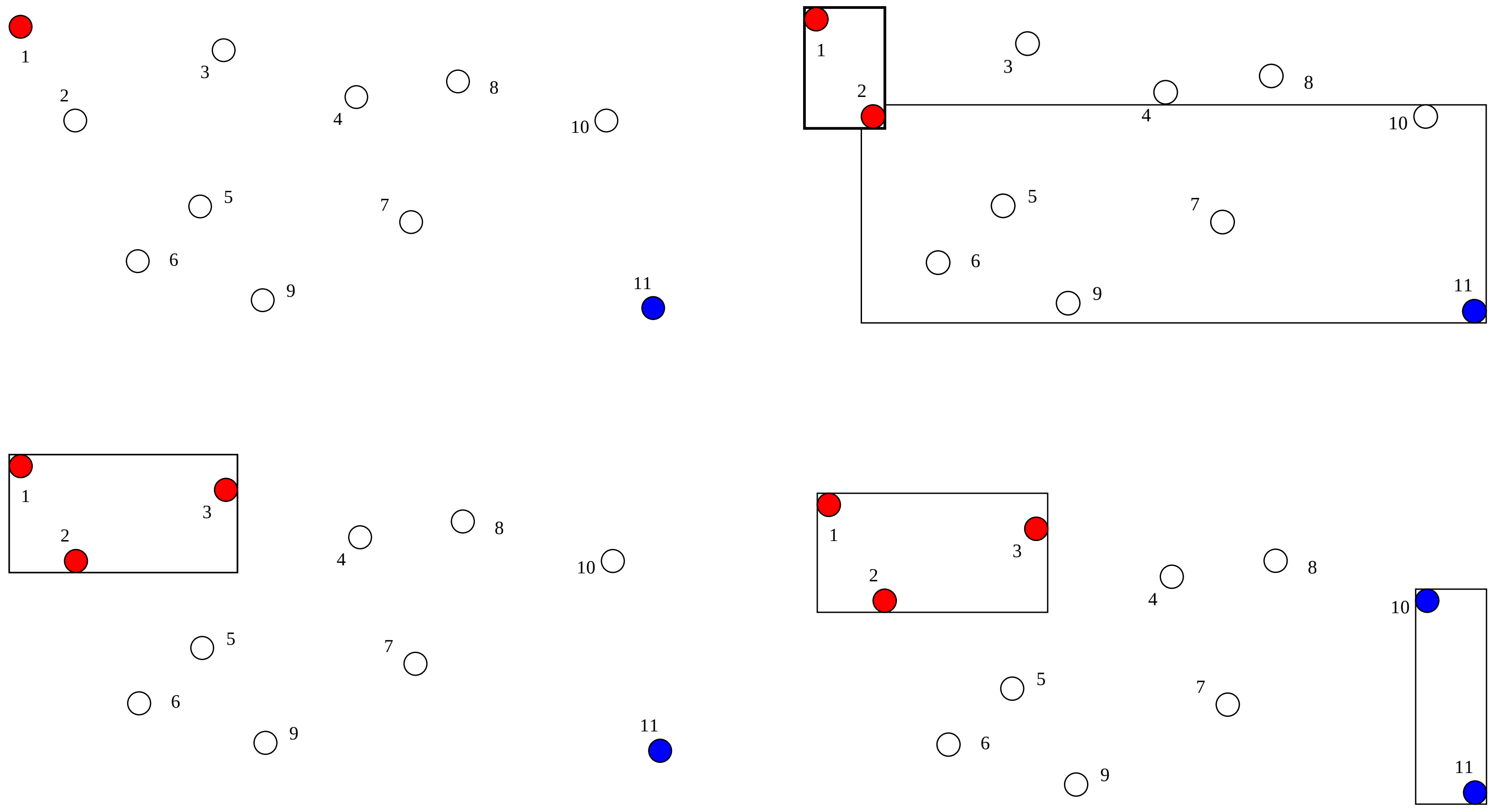
- PickSeeds

1. For each pair of entries E1 and E2, compose a rectangle J including E1 and E2. Calculate d = area(J) - area(E1) - area(E2)
2. Choose the pair with the largest d (d is the dead space that contains nothing)

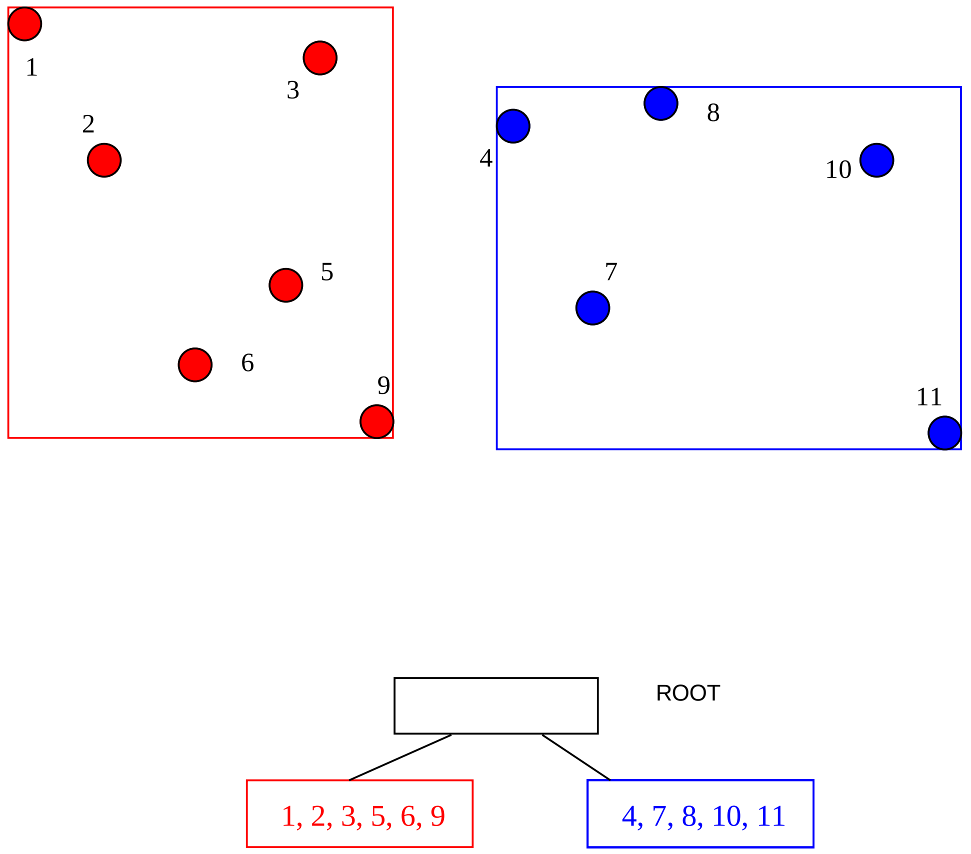
- PickNext

1. For each entry E not yet in a group, calculate d1=the area increase required in the covering rectangle of Group 1 to include E. Calculate d2 similarly for Group 2
2. Choose any entry with the maximum difference between d1 and d2

Split Example (maximum capacity 10, minimum 4)



Nodes after split



R-tree insertion

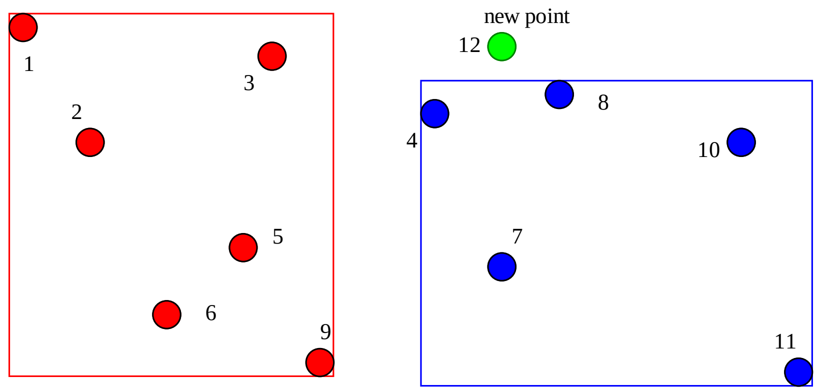
1. Invoke ChooseSubtree to find an appropriate node N, which to place the new object
2. If N has < *b* entries, add object in N (no overflow)
3. If N is already full, Split N and propagate split upwards
4. Adjust all covering rectangles in the insertion path such that they are the MBRs enclosing their children rectangles

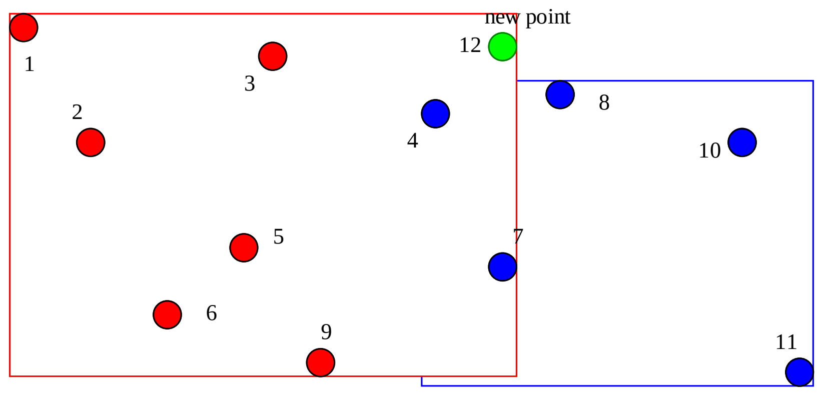
- Algorithm ChooseSubtree

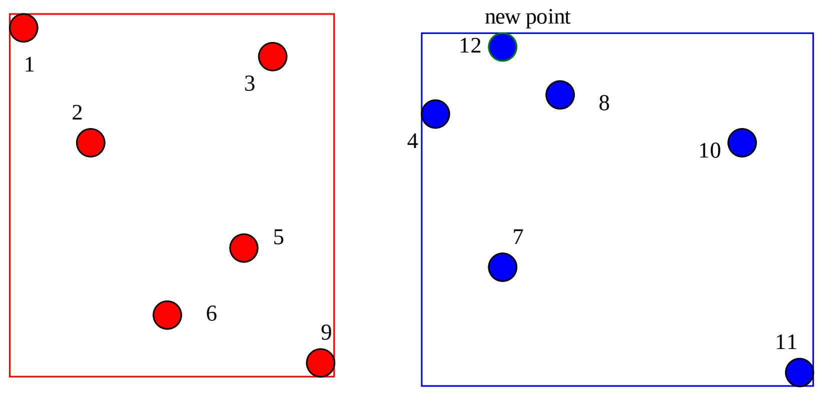
1. Set N to be the root
2. If N is a leaf, return N
3. else choose the entry of N that incurs the minimum MBR increase. Resolve ties by choosing the entry with the with the smallest MBR
4. Set N to be the childnode pointed to by the pointer of the chosen entry and goto 2

5. Deletion: Find the object to be deleted using a top down search of the R-tree and delete it. If there is an **underflow** (i.e., after deletion the node contains < m entries), delete and re-insert all the objects in the node (there is no re-distribution or merging like the B+-tree).

Example of ChooseSubtree

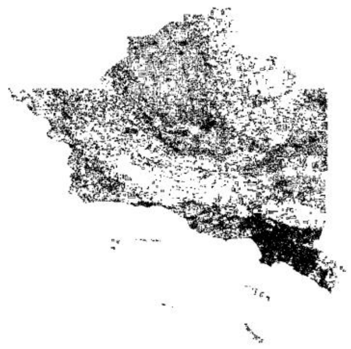


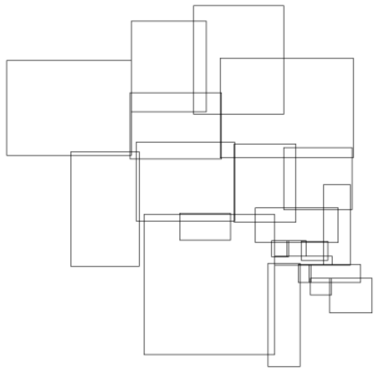


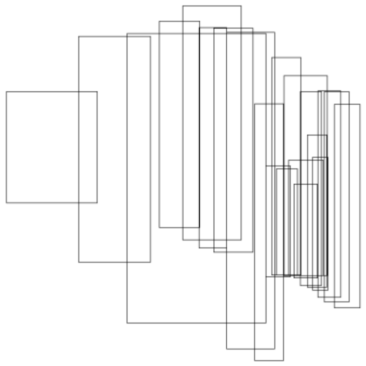


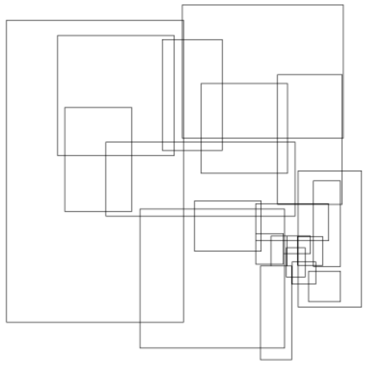
Bulk-loaded R-trees

- For static objects: Sort objects according to geometric criterion (e.g., x-coordinates, Hilbert value) and insert objects in nodes according to the sorted order (i.e., first *b* objects go to the first leaf node and so on).
- Much faster than individual insertions. Trees are packed (i.e., nodes are full)
-

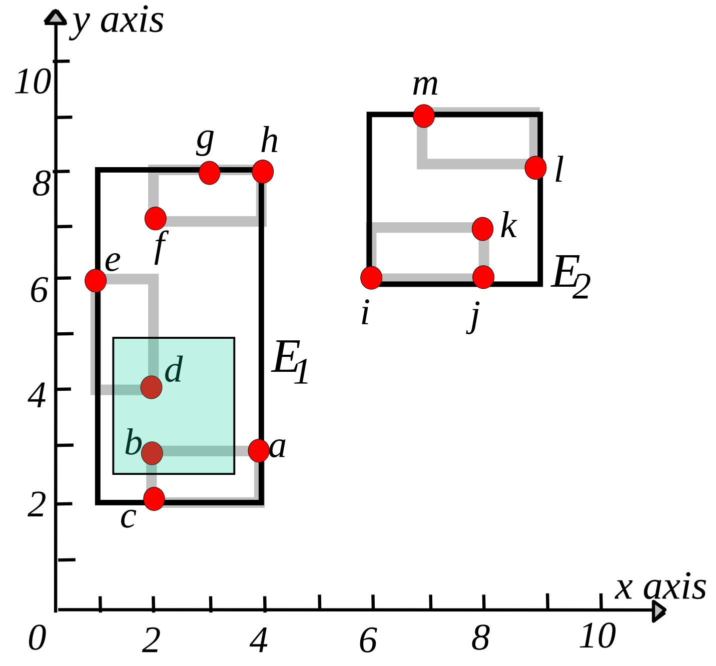

(a) California roads

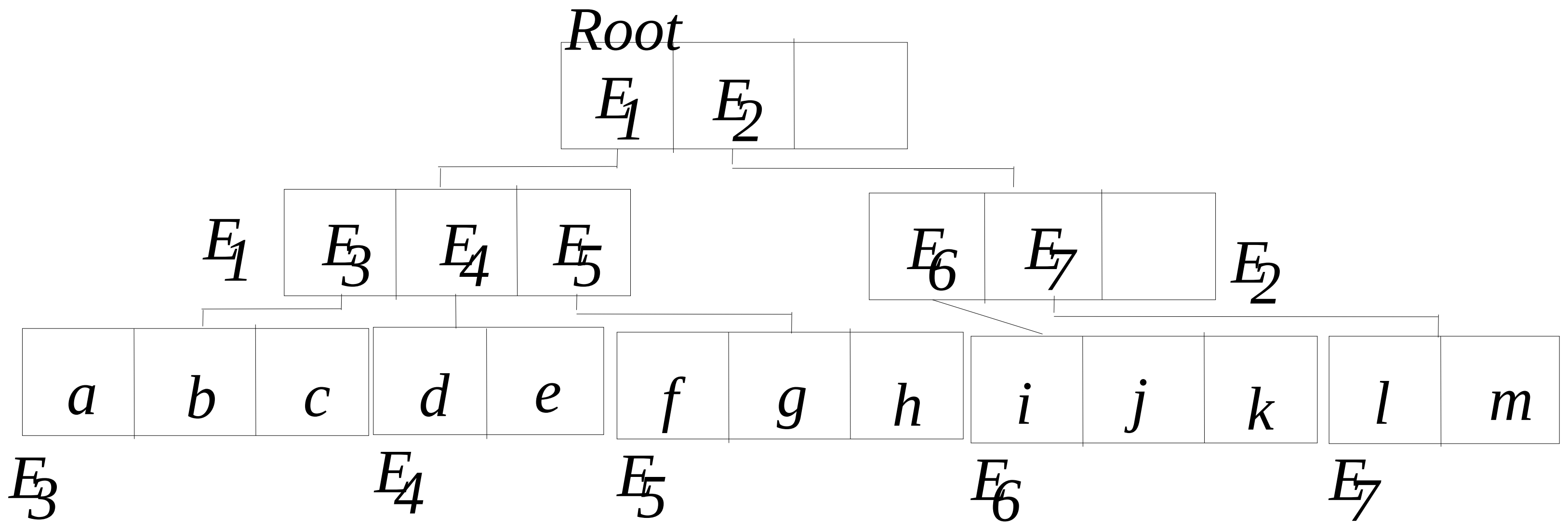

(b) Node MBRs of R* -tree


(c) Sorting on x-coordinate

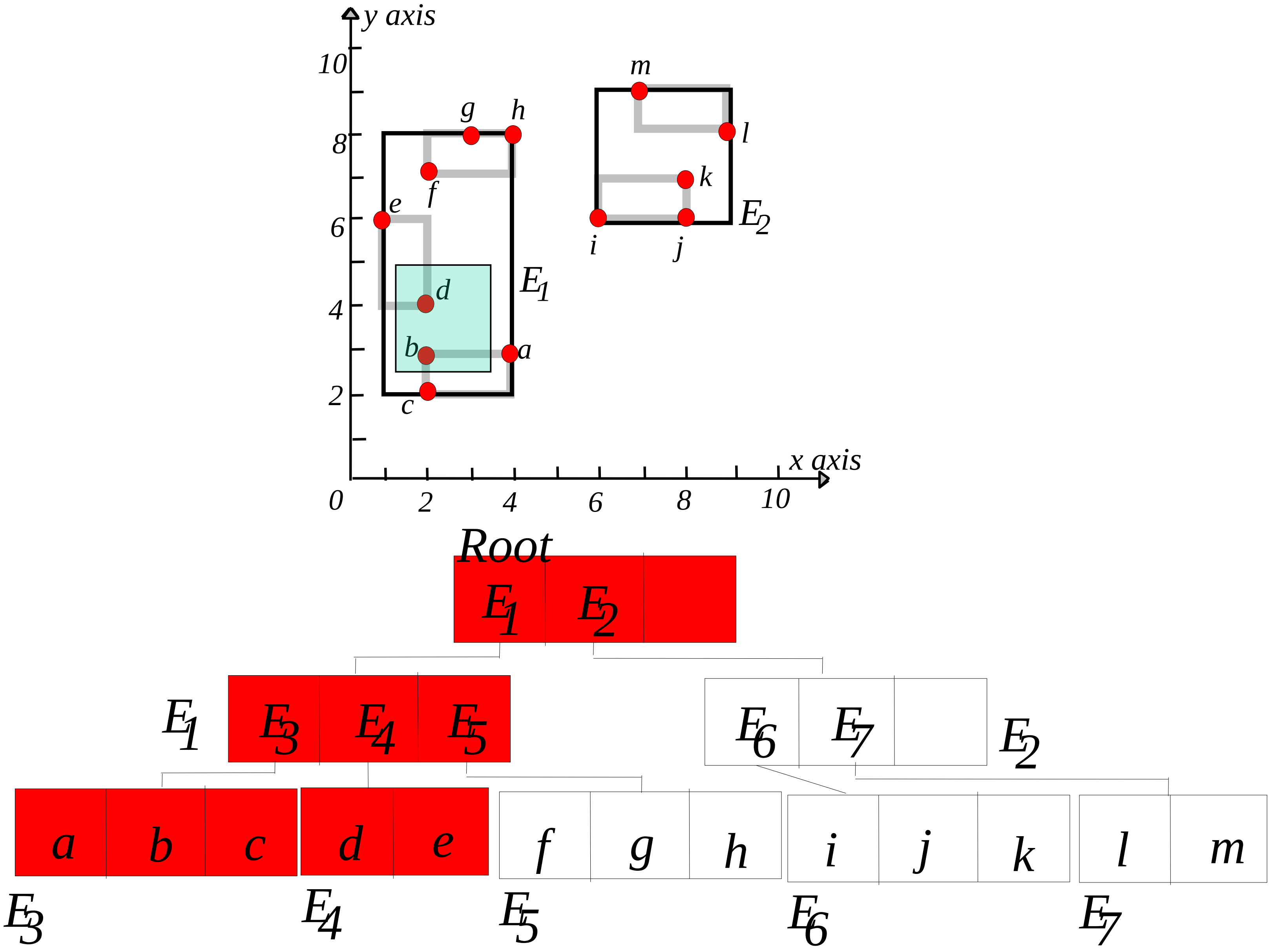

(d) Sorting on Hilbert value

R-tree, Range Query



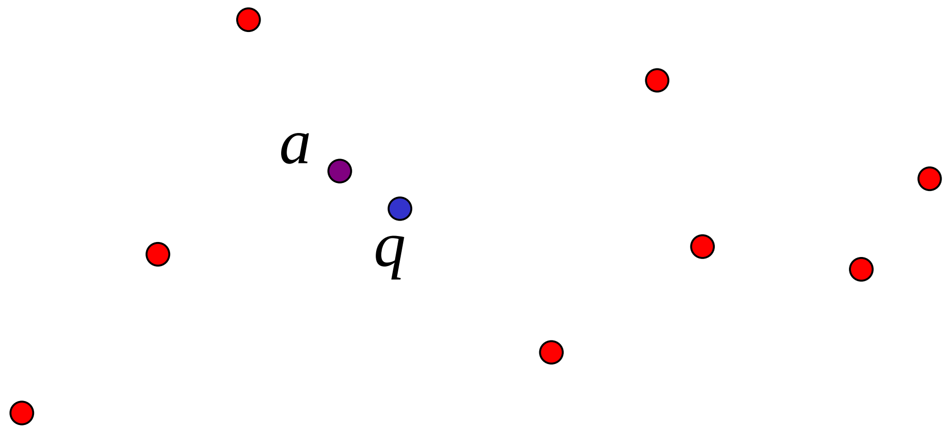


Range Query


y axis
x axis
0
2
4
6
8
10
a
b
c
d
e
f
g
h
i
j
k
l
m
E_1
E_2
Root
E_1 E_2
E_1
E_3 E_4 E_5
E_6 E_7
E_2
a b c
d e
f g h
i j k
l m
E_3
E_4
E_5
E_6
E_7

Nearest Neighbor (NN) Query

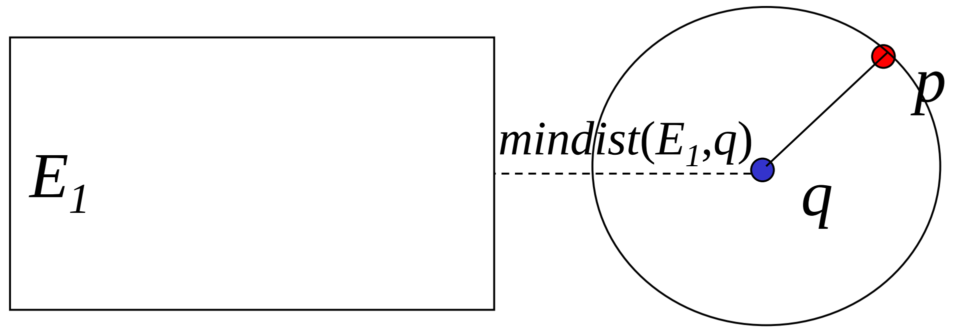
- Given a query location *q*, find the nearest object.



- Depth First and Best-First Search using R-trees
- Our goal: avoid visiting nodes that cannot contain results

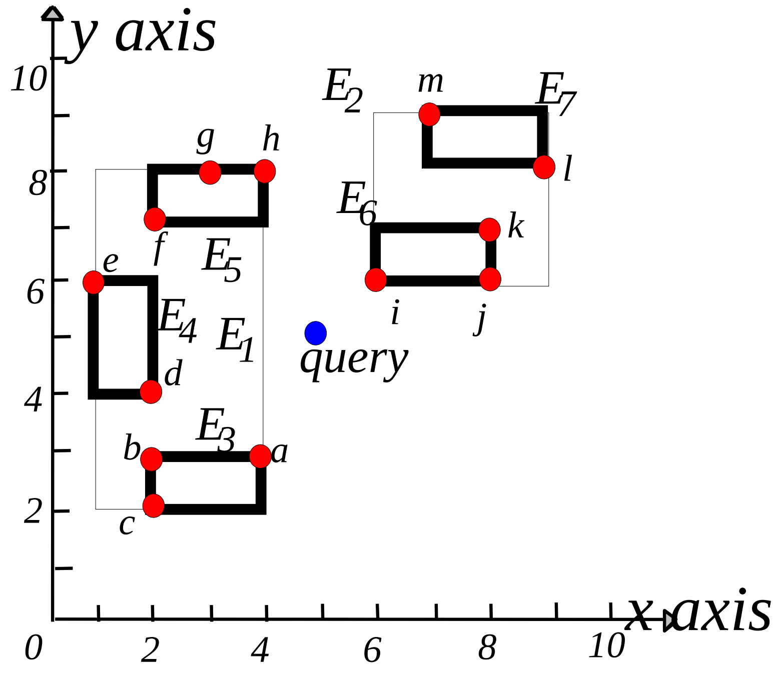
Basic Pruning Metric: MINDIST

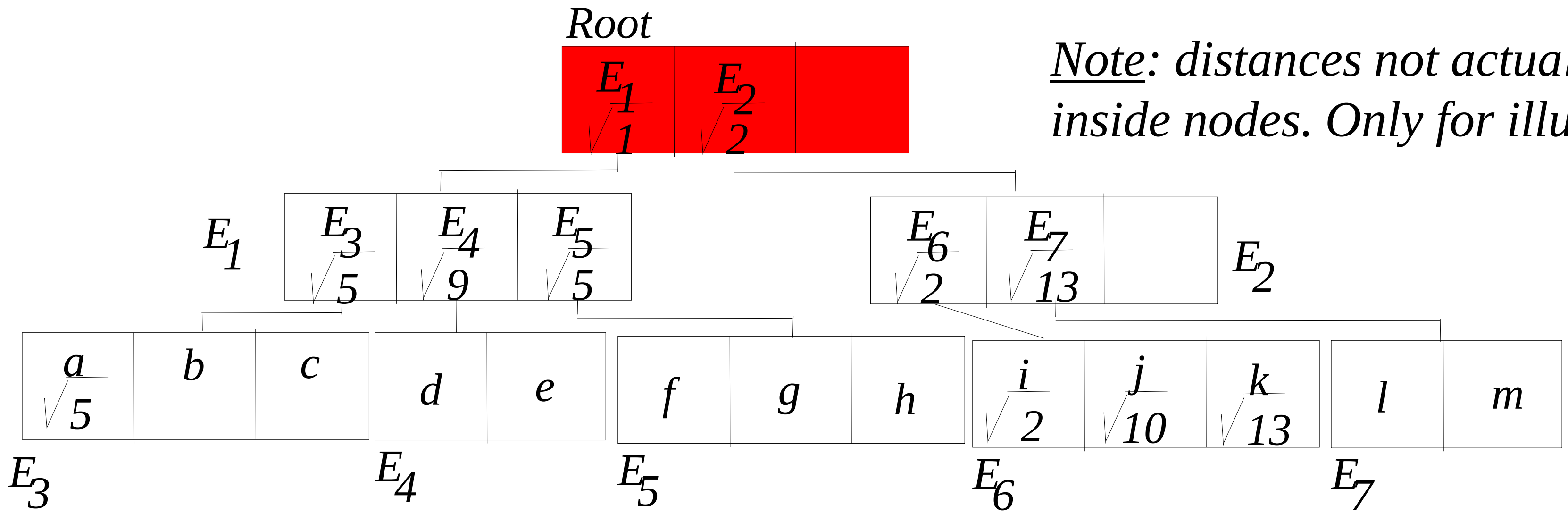
- Minimum distance between *q* and an MBR.



- *mindist*(E_1,*q*) is a lower bound of *d*(*o*,*q*) for every object *o* in E_1.
- If we have found a candidate NN *p*, we can prune every MBR whose *mindist* > *d*(*p*, *q*).

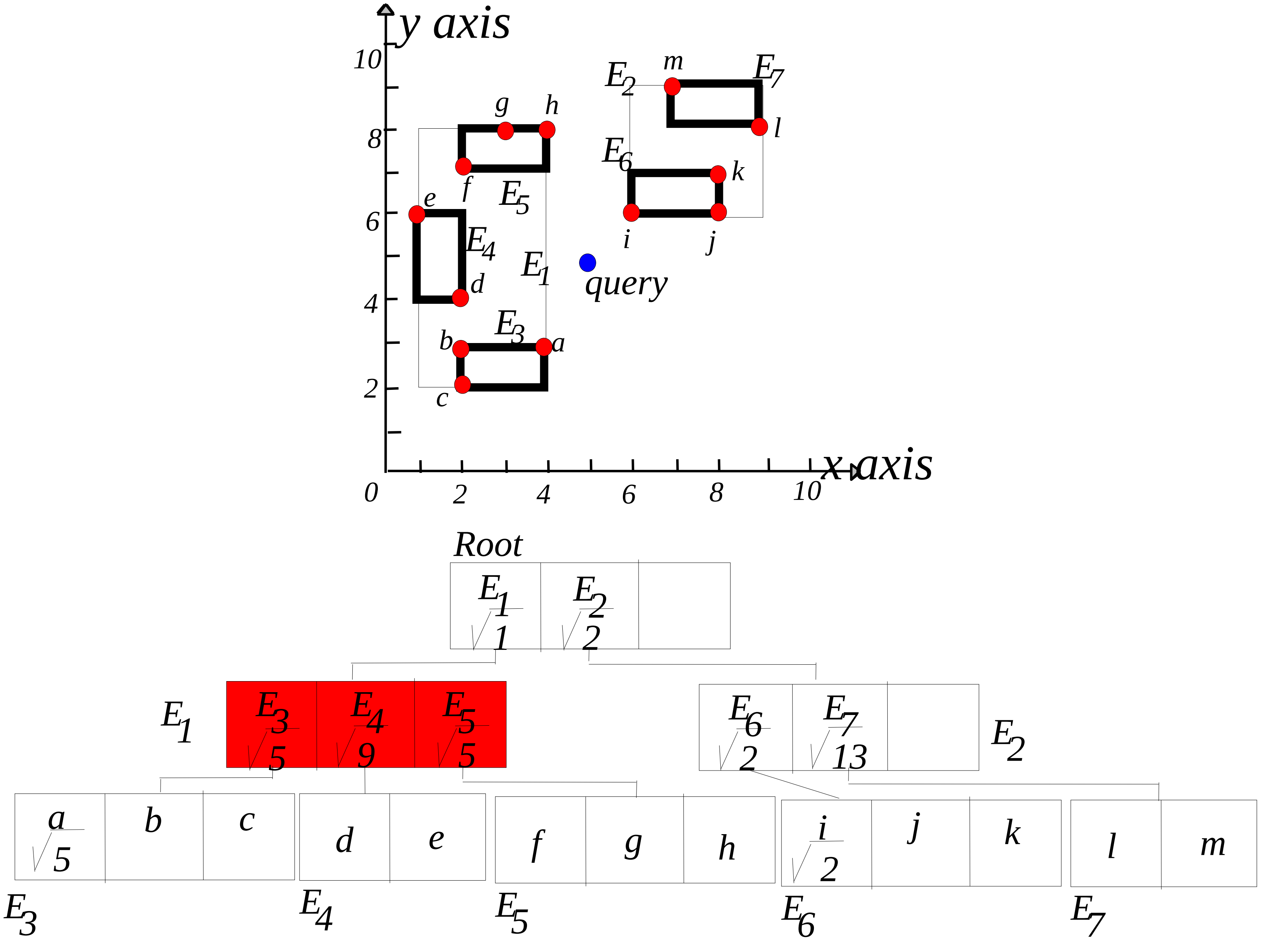
Depth-First (DF) NN Algorithm



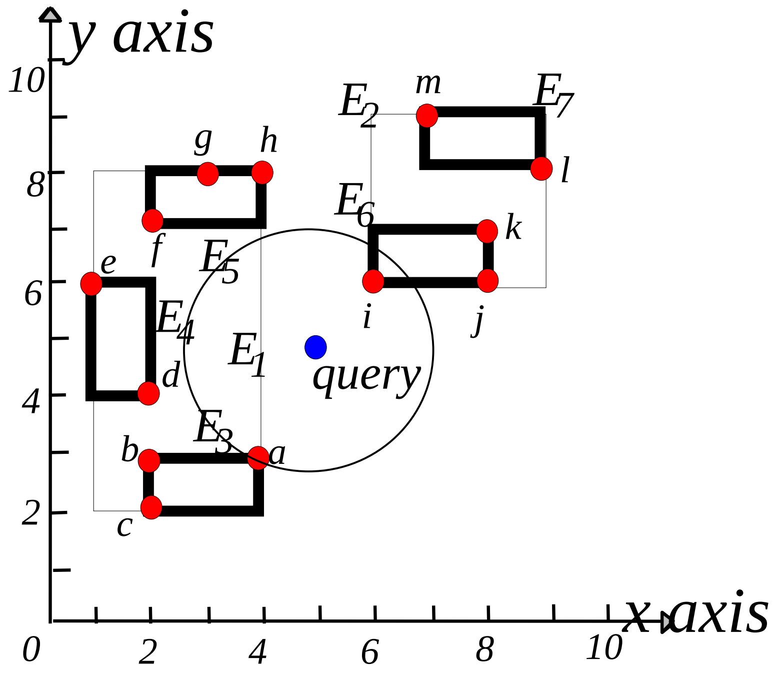


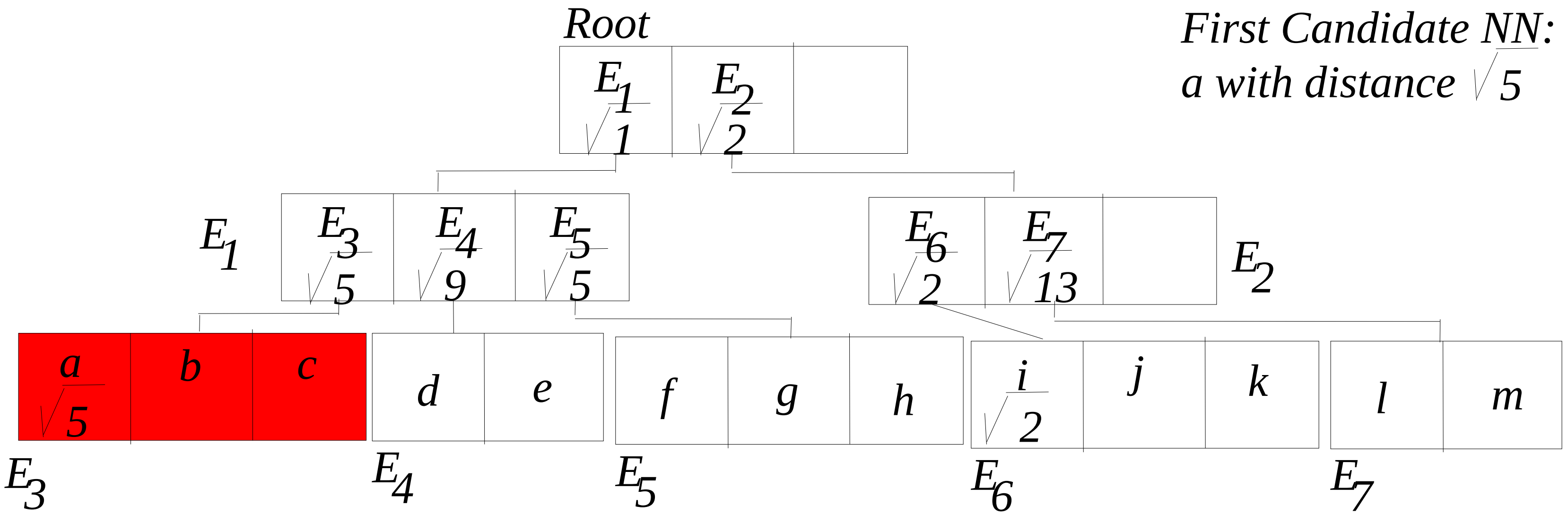
Note: distances not actually stored inside nodes. Only for illustration

DF Search – Visit E_1



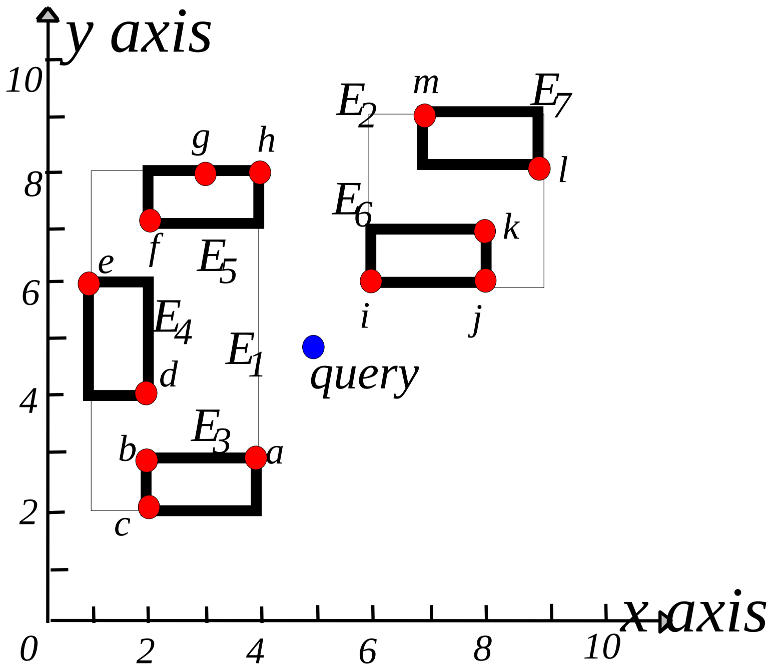
DF Search – Find Candidate NN *a*

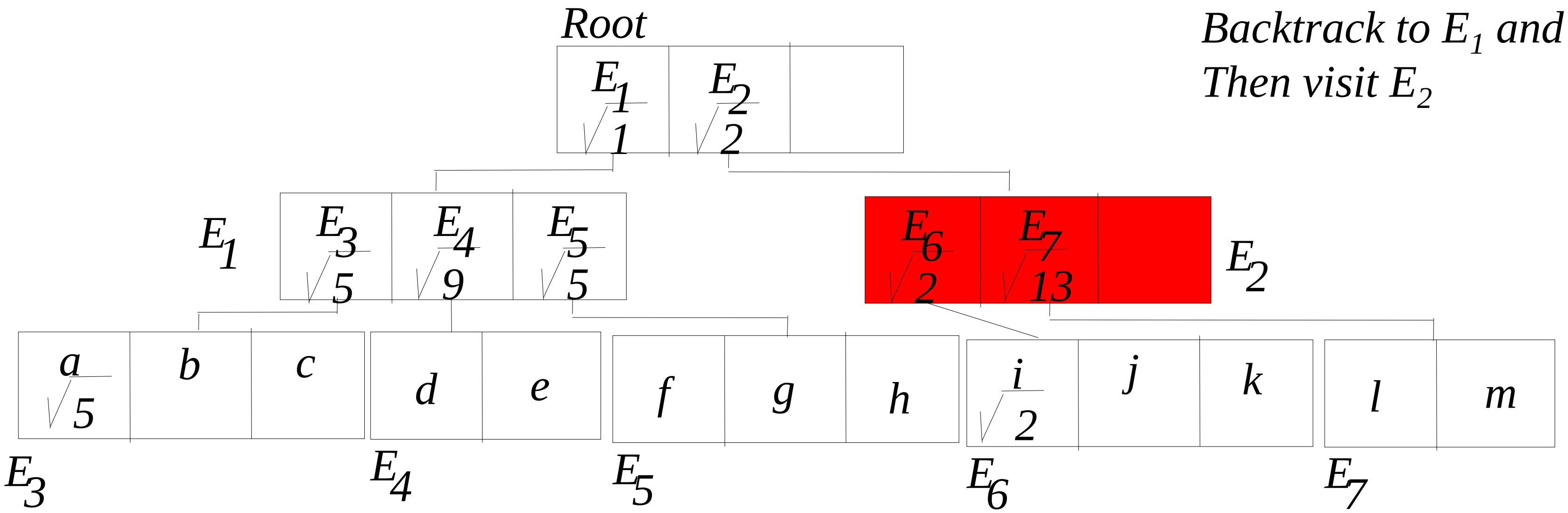




First Candidate NN:
a with distance $\sqrt{5}$

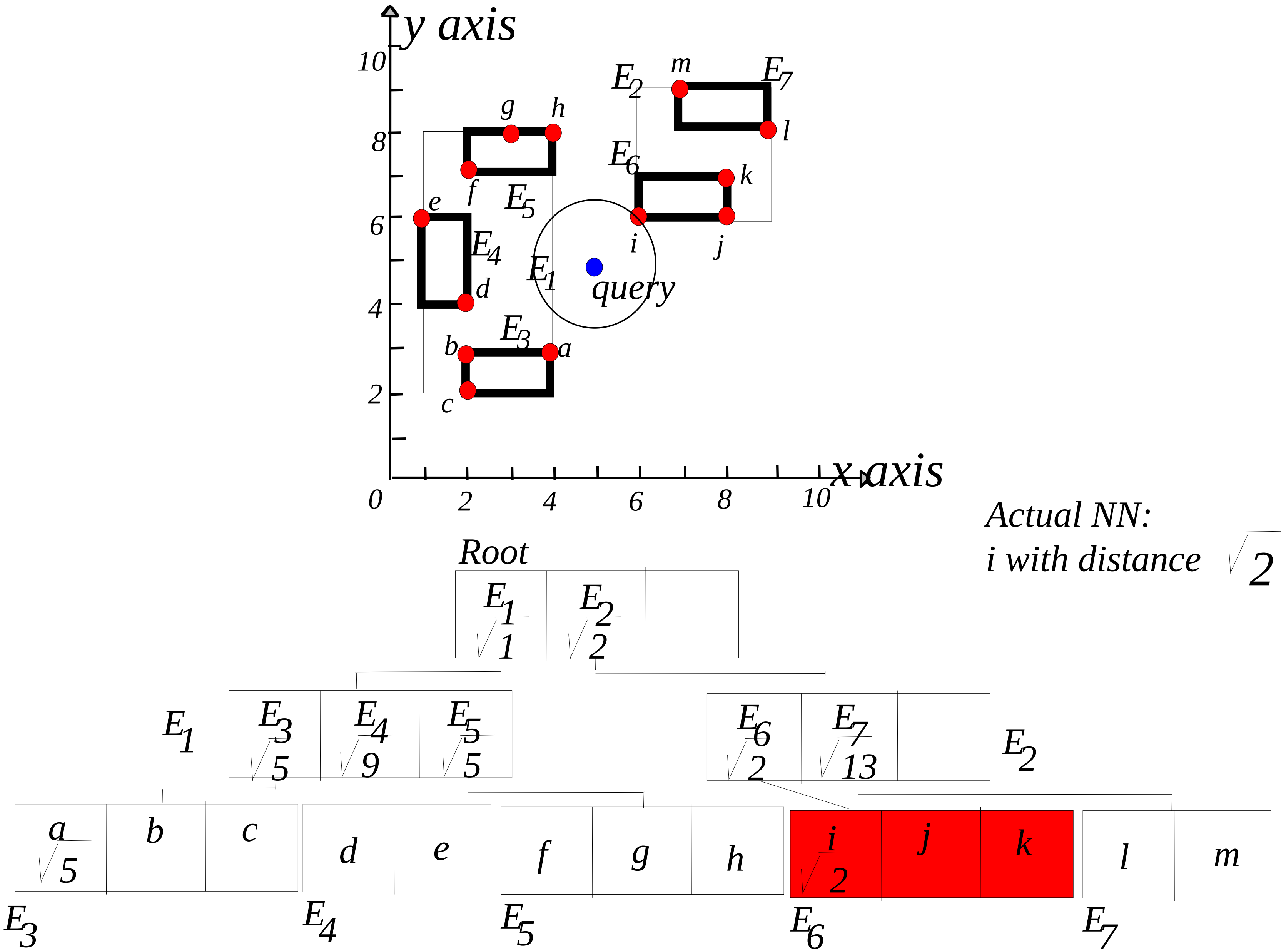
Backtrack to Root and Visit E_2





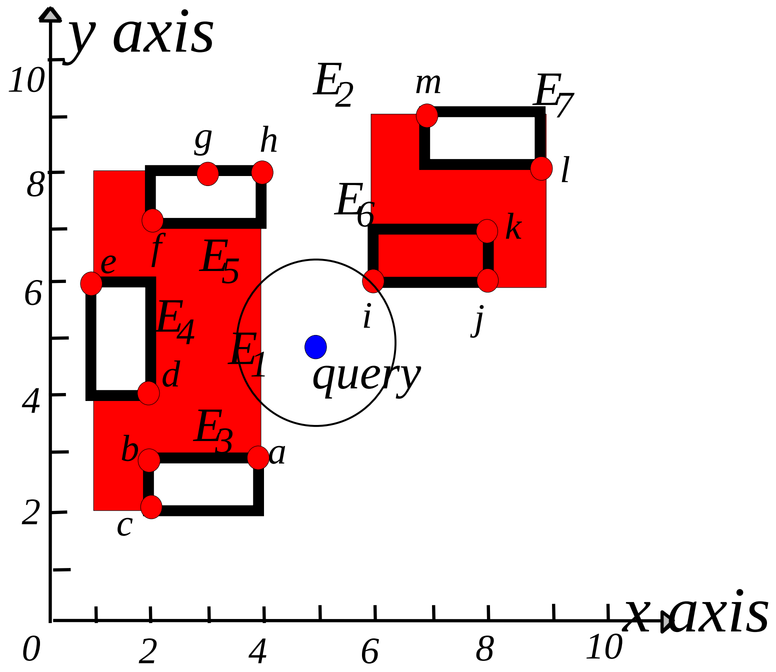
Backtrack to E_1 and Root
Then visit E_2

DF Search – Find Actual NN *i*



Actual NN:
i with distance $\sqrt{2}$

Optimality

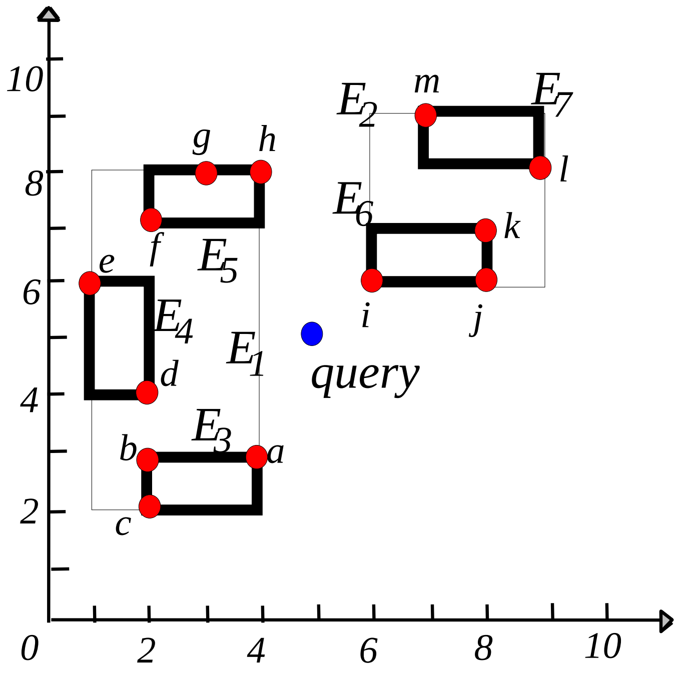


- Question: Which is the minimal set of nodes that must be visited by any NN algorithm?
- Answer: The set of nodes whose MINDIST is smaller than or equal to the distance between *q* and its NN (e.g., E_1, E_2, E_6).

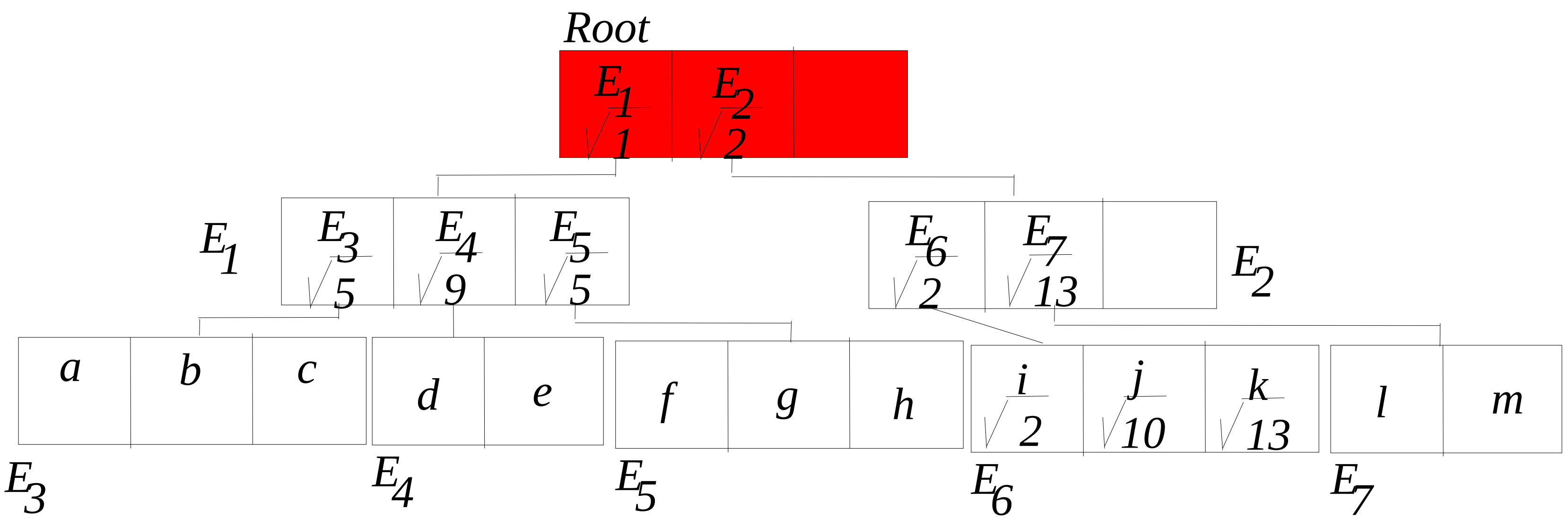
Best-First (BF) NN Algorithm (Optimal)

- Keep a heap *H* of index entries and objects, ordered by MINDIST.
- Initially, *H* contains the root.
- While $H \neq \phi$
  - Extract the element with minimum MINDIST
    - If it is an index entry, insert its children into *H*.
    - If it is an object, return it as NN.
- End while

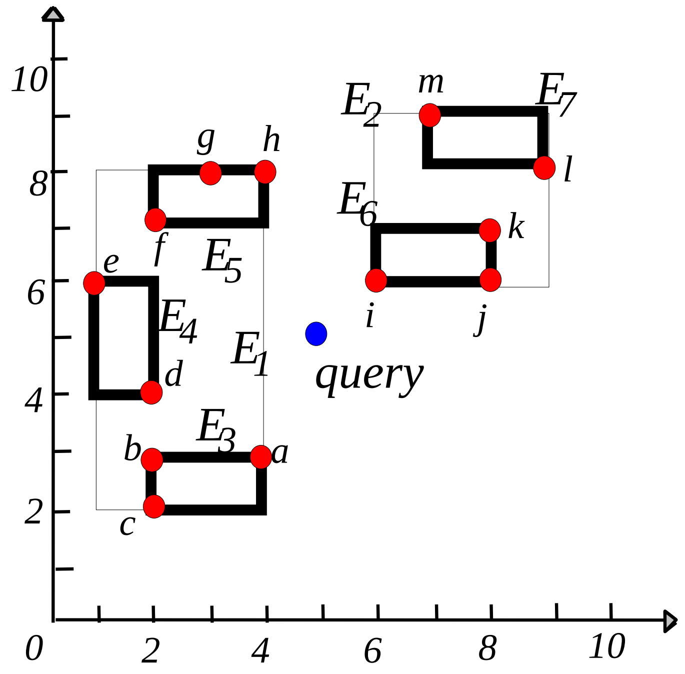
BF Search – Visit *root*



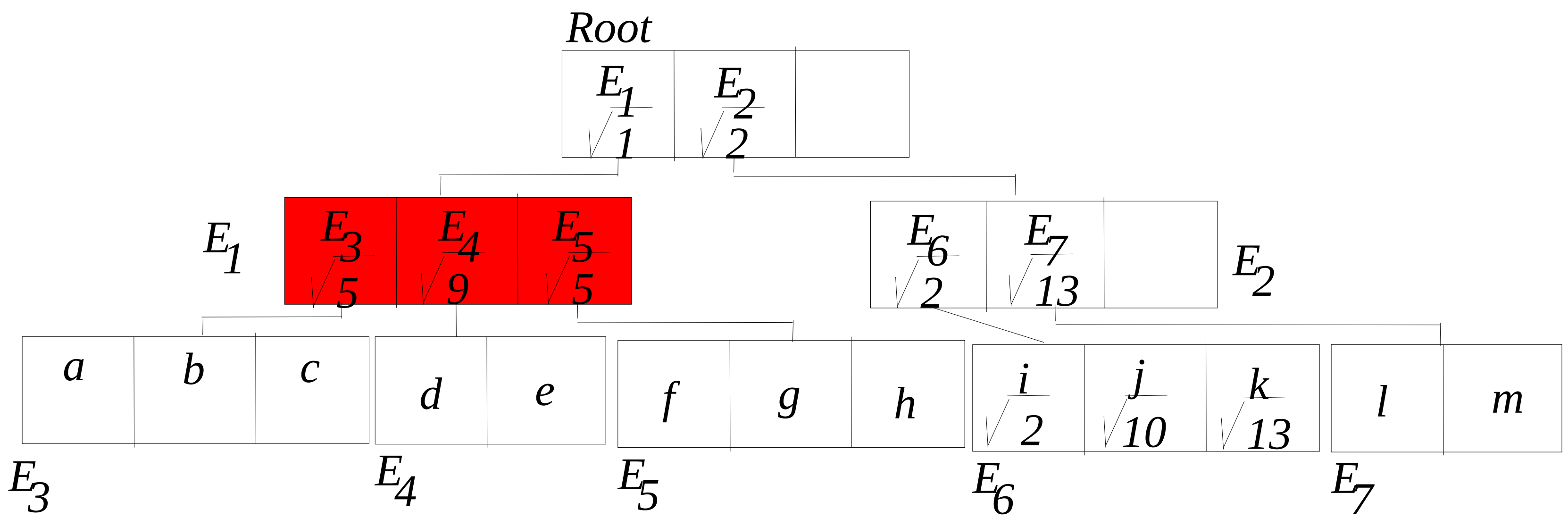
Action	*Heap*						
Visit Root	E_1 $\sqrt{1}$	E_2 $\sqrt{2}$					



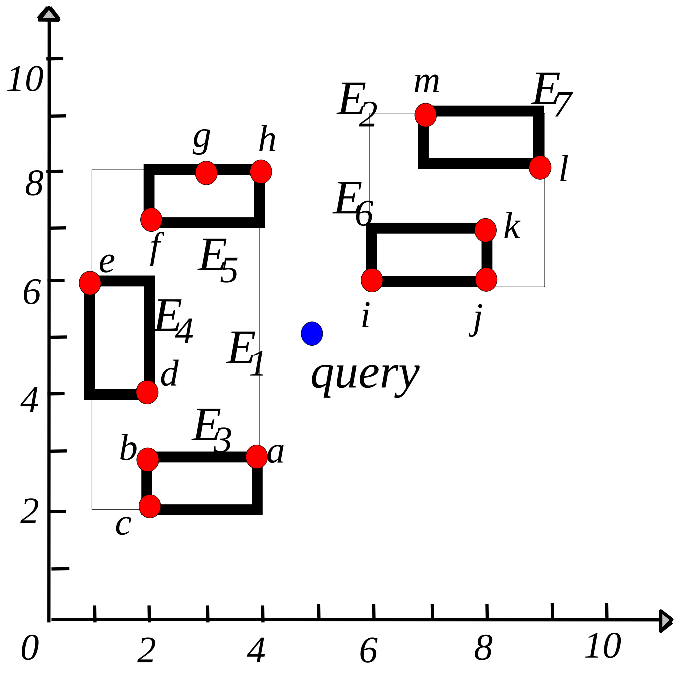
BF Search – Visit E_1



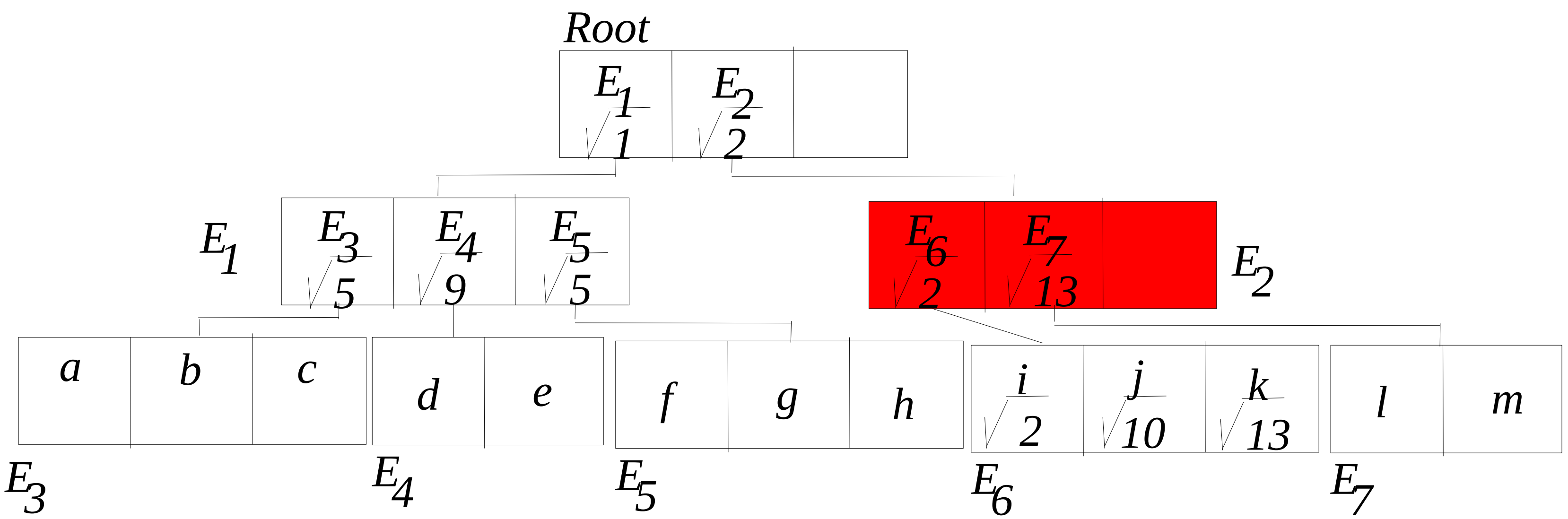
Action	Heap						
Visit Root	$E_1\ \sqrt{1}$	$E_2\ \sqrt{2}$					
follow E_1	$E_2\ \sqrt{2}$	$E_3\ \sqrt{5}$	$E_5\ \sqrt{5}$	$E_4\ \sqrt{9}$			



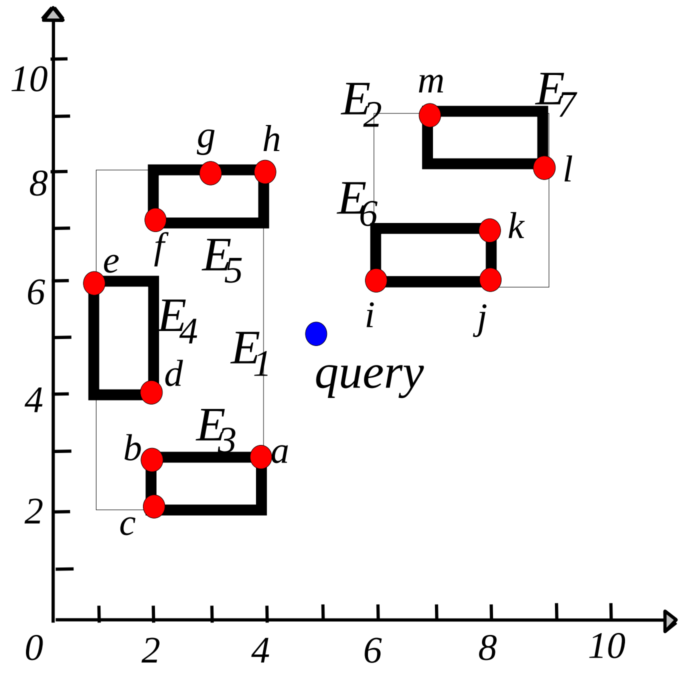
BF Search – Visit E_2



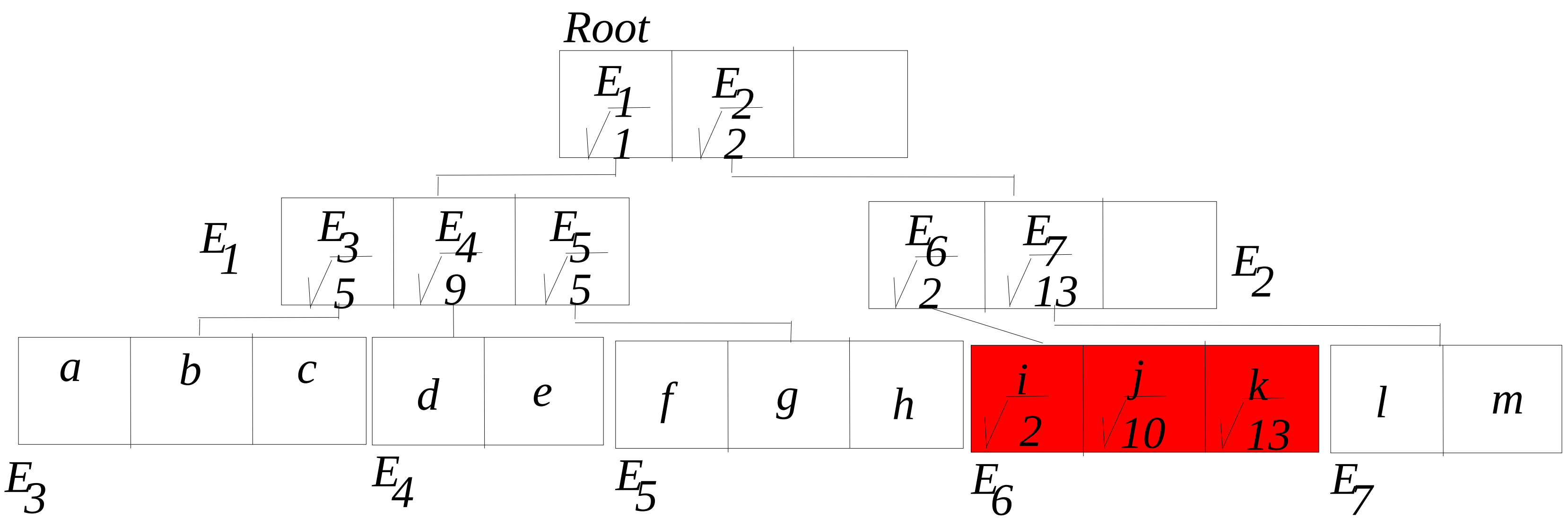
Action		Heap					
Visit Root	$E_1\ \sqrt{1}$	$E_2\ \sqrt{2}$					
follow E_1	$E_2\ \sqrt{2}$	$E_3\ \sqrt{5}$	$E_5\ \sqrt{5}$	$E_4\ \sqrt{9}$			
follow E_2	$E_6\ \sqrt{2}$	$E_3\ \sqrt{5}$	$E_5\ \sqrt{5}$	$E_4\ \sqrt{9}$	$E_7\ \sqrt{13}$		



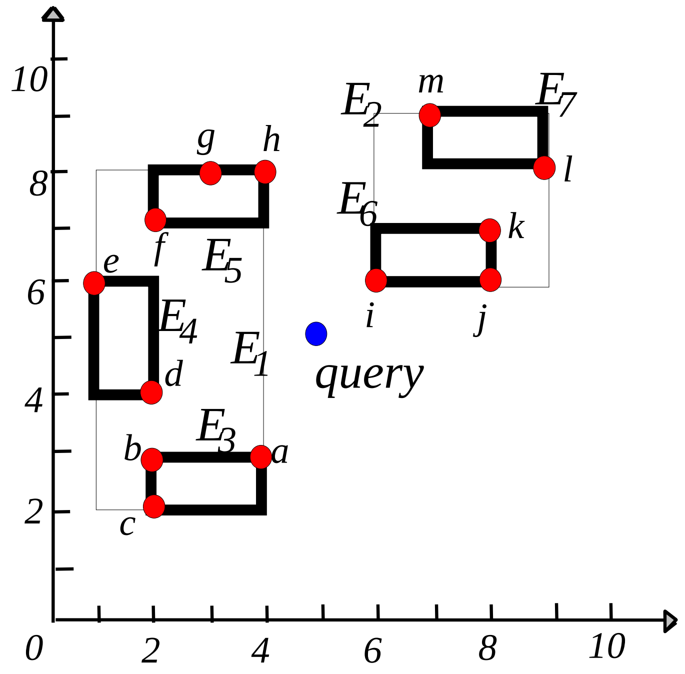
BF Search – Visit E_6



Action	Heap						
Visit Root	E_1 $\sqrt{1}$	E_2 $\sqrt{2}$					
follow E_1	E_2 $\sqrt{2}$	E_3 $\sqrt{5}$	E_5 $\sqrt{5}$	E_4 $\sqrt{9}$			
follow E_2	E_6 $\sqrt{2}$	E_3 $\sqrt{5}$	E_5 $\sqrt{5}$	E_4 $\sqrt{9}$	E_7 $\sqrt{13}$		
follow E_6	i $\sqrt{2}$	E_3 $\sqrt{5}$	E_5 $\sqrt{5}$	E_4 $\sqrt{9}$	j $\sqrt{10}$	E_7 $\sqrt{13}$	k $\sqrt{13}$

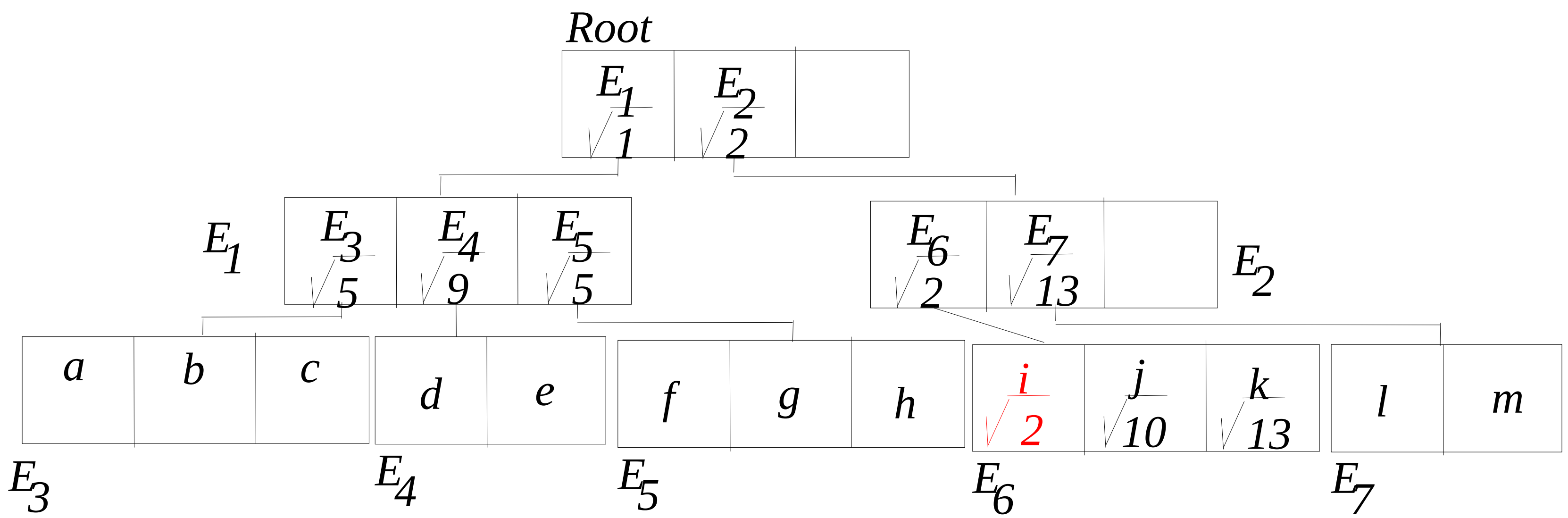


DF Search – Find Actual NN *i*



Action	Heap						
Visit Root	E_1 $\sqrt{1}$	E_2 $\sqrt{2}$					
follow E_1	E_2 $\sqrt{2}$	E_3 $\sqrt{5}$	E_5 $\sqrt{5}$	E_4 $\sqrt{9}$			
follow E_2	E_6 $\sqrt{2}$	E_3 $\sqrt{5}$	E_5 $\sqrt{5}$	E_4 $\sqrt{9}$	E_7 $\sqrt{13}$		
follow E_6	i $\sqrt{2}$	E_3 $\sqrt{5}$	E_5 $\sqrt{5}$	E_4 $\sqrt{9}$	j $\sqrt{10}$	E_7 $\sqrt{13}$	k $\sqrt{13}$

Report i and terminate

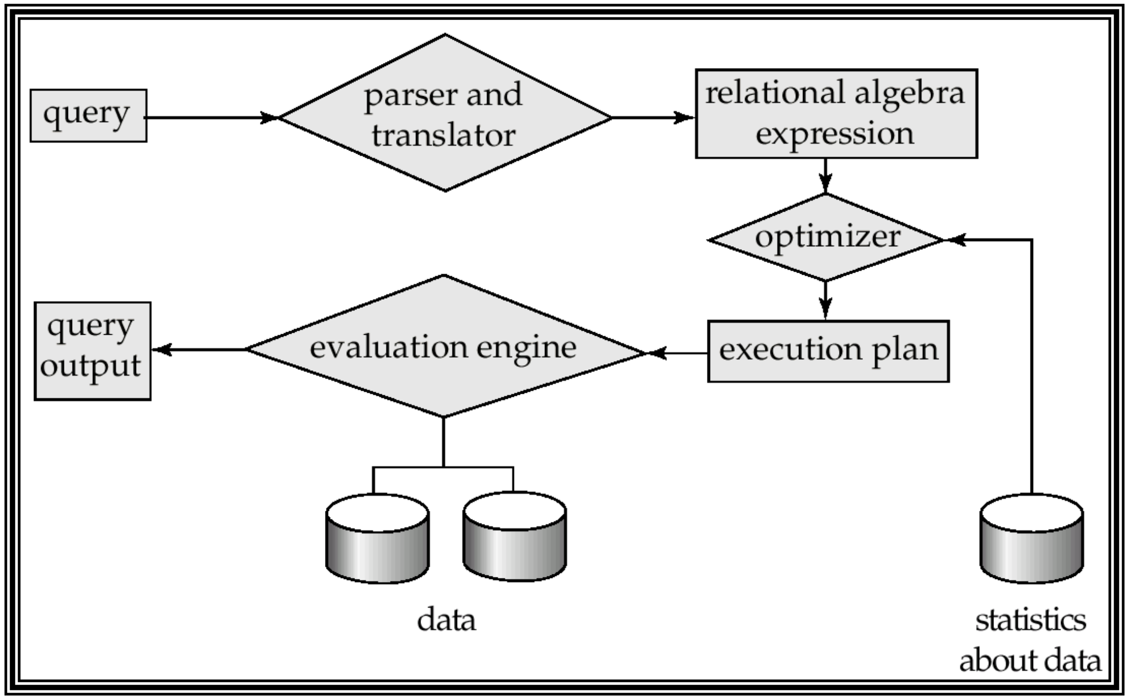


Discussion about NN Queries

- Both DF and BF can be easily adapted to (i) extended (instead of point) objects and (ii) retrieval of *k* (>1) NN.
- BF is incremental; i.e., it can report the NN in increasing order of distance without a given value of *k*.
  - Example: find the 10 closest cities to HK with population more than 1 million. We may have to retrieve many (>>10) cities around Hong Kong in order to answer the query.

Basic Steps in Query Processing

- **Parsing and translation**
  - Query is checked for syntax, relations are verified and query is translated into relational algebra.
- **Evaluation**
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.



Basic Steps in Query Processing : Optimization

- A relational algebra expression may have many equivalent expressions
  - E.g., $\sigma_{balance<2500}(\Pi_{balance}(account))$ is equivalent to
    $\Pi_{\text{balance}}(\sigma_{\text{balance}<2500}(account))$
- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**.
  - E.g., can use an index on *balance* to find accounts with balance < 2500,
  - or can perform complete relation scan and discard accounts with balance $\geq$ 2500

Basic Steps: Optimization (Cont.)

- **Query Optimization**: Amongst all equivalent evaluation plans choose the one with the (expected) lowest cost.
  - Cost is estimated using statistical information from the database catalog, e.g. number of tuples in each relation, size of tuples, etc.
- For simplicity we just use *number of page transfers from disk* as the cost measure
  - We ignore the difference in cost between sequential and random I/O for simplicity. We also ignore CPU costs for simplicity
- Costs depends on the size of the buffer in main memory
  - Having more memory reduces need for disk access
  - Amount of real memory available to buffer depends on other concurrent OS processes, and hard to determine
  - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available
- Real systems take CPU cost into account, differentiate between sequential and random I/O, and take buffer size into account
- We do not include cost of writing the final output to disk

Selection Operation

- **File scan** – search algorithms that locate and retrieve records that fulfill a selection condition.
- Algorithm **A1** (*linear search*). Scan each page and test all records to see whether they satisfy the selection condition.
  - Cost estimate (number of disk pages scanned) = b_r
    - b_r denotes number of pages containing records from relation r
  - If selection is on a key attribute, cost = $(b_r/2)$
    - stop on finding record
  - Linear search can be applied regardless of
    - selection condition or
    - ordering of records in the file, or
    - availability of indices

Selection Operation (Cont.)

- **A2** *(binary search).* Applicable if selection is an equality comparison on the attribute on which file is ordered.
  - Assume that the pages of a relation are stored contiguously
  - Cost estimate (number of disk pages to be scanned):
    - $\lceil \log_2(b_r) \rceil$ — cost of locating the first tuple by a binary search on the pages
    - *Plus* number of pages containing records that satisfy selection condition

Selections Using Indices

- **A3** (*primary index on candidate key, equality*). Retrieve a single record that satisfies the corresponding equality condition
  - *Cost* = HT_i + 1 (HT_i is the height of the tree index - if hash index HT_i = 1 or HT_i = 1.2 if we assume that there exist overflow buckets)
- **A4** (*primary index on nonkey, equality)* Retrieve multiple records.
  - Records will be on consecutive pages
  - *Cost* = HT_i + number of pages containing retrieved records
- **A5** (*equality on search-key of secondary index).*
  - Retrieve a single record if the search-key is a candidate key
    - *Cost* = HT_i + 1
  - Retrieve multiple records if search-key is not a candidate key
    - Cost = HT_i + *number of records retrieved*
      - Can be very expensive!
    - each record may be on a different page
      - one page access for each retrieved record

Selections Involving Comparisons

Can implement selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$ by using

- a linear file scan or binary search,
- or by using indices in the following ways:

- **A6** (*primary index, comparison).* (Relation is sorted on A)
  - For $\sigma_{A \geq V}(r)$ use index to find first tuple $\geq v$ and scan relation sequentially from there
  - For $\sigma_{A \leq V}(r)$ just scan relation sequentially till first tuple $> v$; do not use index
- **A7** (*secondary index, comparison*).
  - For $\sigma_{A \geq V}(r)$ use index to find first index entry $\geq v$ and scan index sequentially from there, to find pointers to records.
  - For $\sigma_{A \leq V}(r)$ just scan leaf pages of index finding pointers to records, till first entry $> v$
  - In either case, retrieve records that are pointed to
    - requires an I/O for each record
    - Linear file scan may be cheaper if many records are to be fetched!

Implementation of Complex Selections

Conjunction: $\sigma_{\theta_1 \wedge \theta_2 \wedge \ldots \theta_n}(r)$

- **A8** (*conjunctive selection using one index).*
  - Select a combination of θ_i and algorithms A1 through A7 that results in the least cost for $\sigma_{\theta_i}(r)$.
  - Test other conditions on tuple after fetching it into memory buffer.
- **A9** (*conjunctive selection using multiple-key index*).
  - Use appropriate composite (multiple-key) index if available.
- **A10** (*conjunctive selection by intersection of identifiers).*
  - Requires indices with record pointers.
  - Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers.
  - Then fetch records from file
  - If some conditions do not have appropriate indices, apply test in memory.
  - NOTE: In some cases we do index only scan.

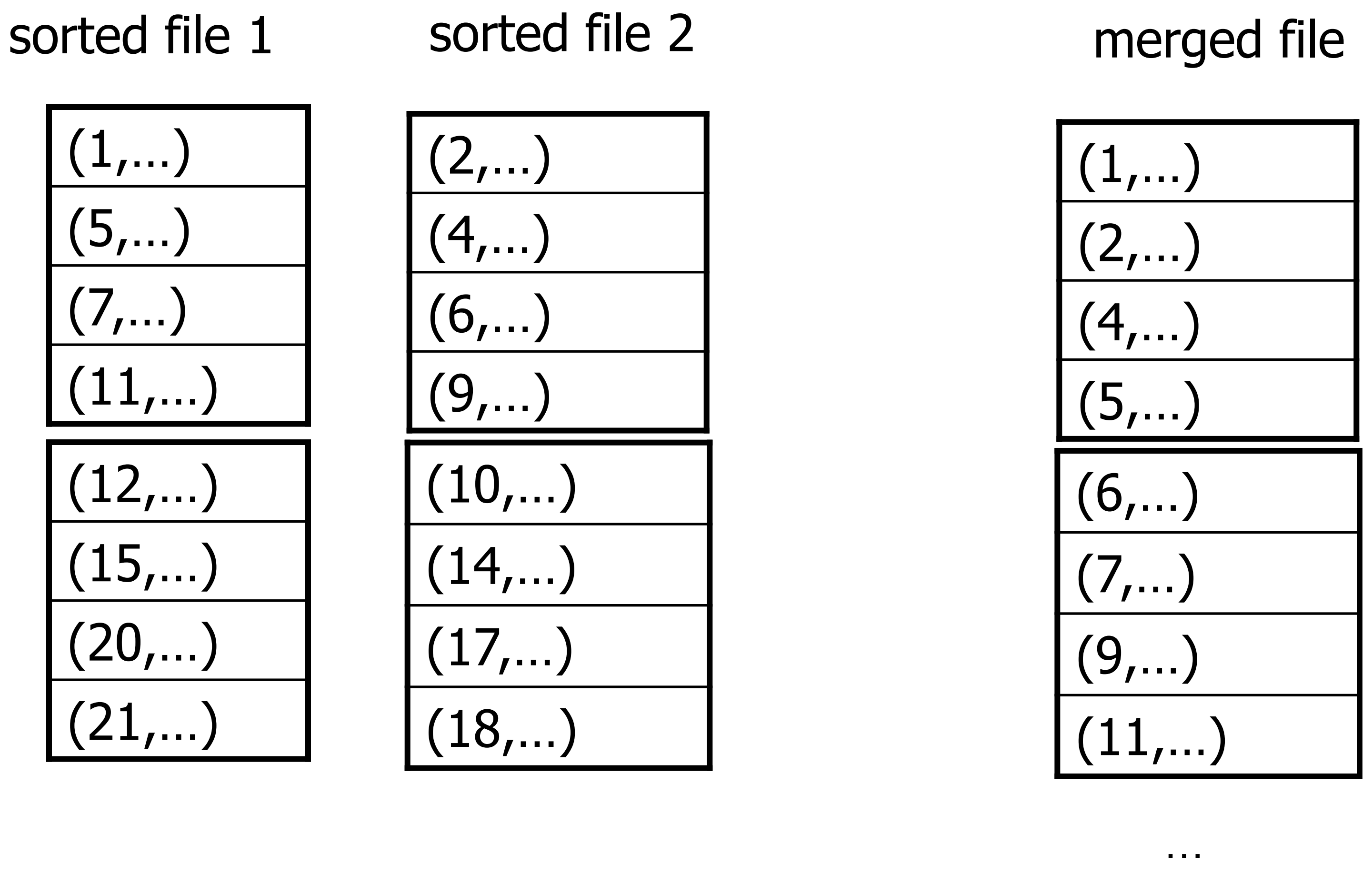
Algorithms for Complex Selections

Disjunction: $\sigma_{\theta_1 \vee \theta_2 \vee \ldots \theta_n}(r)$.

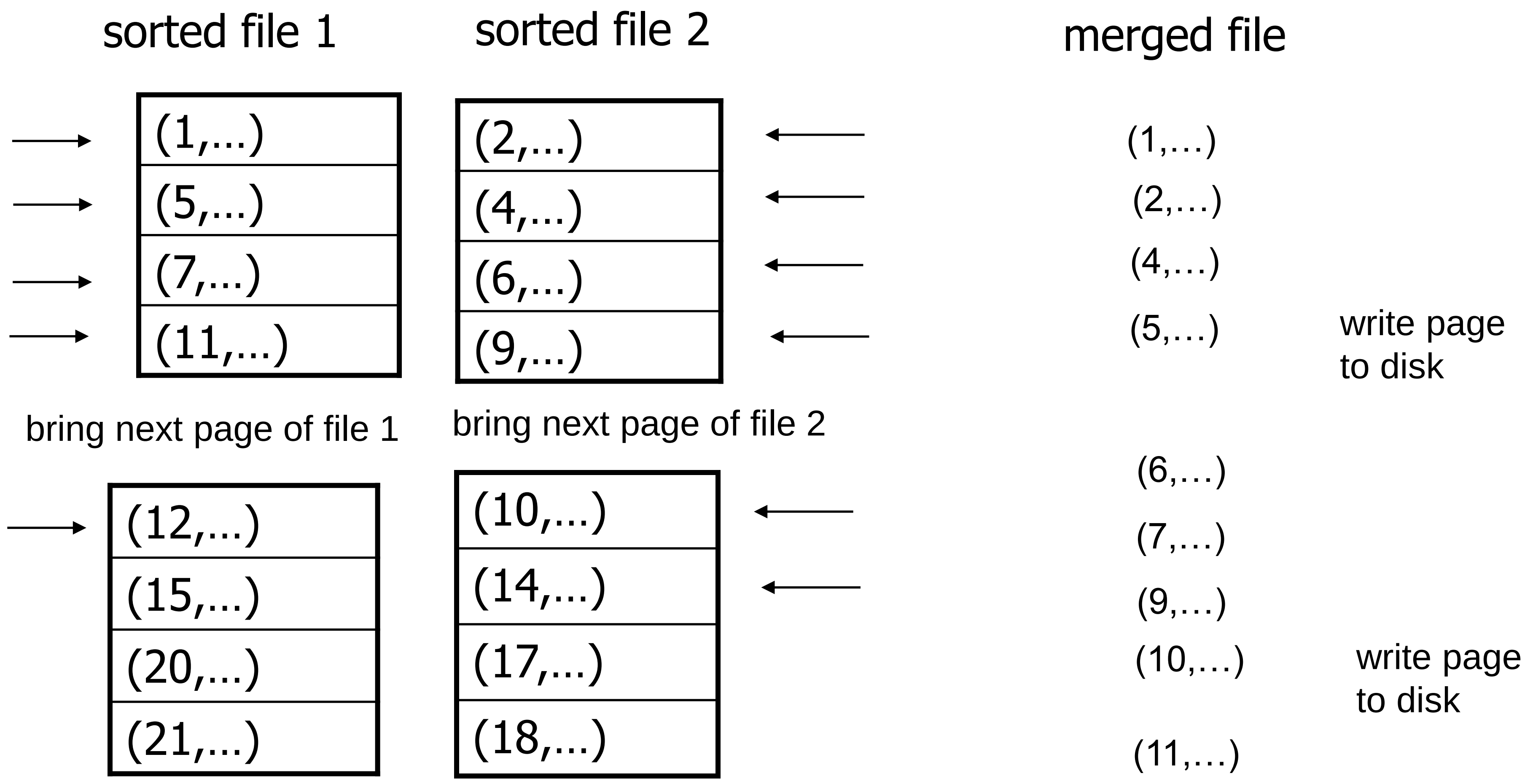
- **A11** (*disjunctive selection by union of identifiers*).
  - Applicable if *all* conditions have available indices.
    - Otherwise use linear scan.
  - Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
  - Then fetch records from file

External Sorting (disk-resident files)

- Merging Sorted Files with 3 pages of main memory buffer



External Sorting (disk-resident files)



External Sorting (disk-resident files)

Continuing the previous example:

Question: I assumed that each file is already sorted. If the file is not sorted, how do I sort it (using only my 3 buffer pages?)

Answer: Each file in the example is only 2 pages. Therefore, I can bring the entire file in memory, and sort it using any main-memory algorithm.

Question: The previous example assumes two separate files. How do I apply this idea to sort a single file?

Answer: You can split the file in two parts and merge them as if they were separate files.

Question: Can I do better if I have $M>3$ main memory pages.

Answer: Yes, instead of 2 you can merge up to M-1 files (because you need 1 page for writing the output).

External Sort-Merge

Let M denote memory size (in pages).

- **Create sorted runs**. Let i be 0 initially.
  Repeatedly the following till the end of the relation:
  (a) Read M pages of relation into memory
  (b) Sort the in-memory pages
  (c) Write sorted data to run R_i; increment i.
  Let the final value of i be N
- **Merge the runs (N-way merge)**. We assume (for now) that $N < M$.

```
Use N pages of memory to buffer input runs, and 1 page to buffer output.
    Read the first page of each run into its buffer page
repeat
    Select the first record (in sort order) among all buffer pages
    Write the record to the output buffer.  If the output buffer is full write it to disk.
    Delete the record from its input buffer page.
        If the buffer page becomes empty then
            read the next page (if any) of the run into the buffer.
until all input buffer pages are empty:
```

External Sort-Merge (Cont.)

- If $i \geq M$, several merge *passes* are required.
  - In each pass, contiguous groups of M - 1 runs are merged.
  - A pass reduces the number of runs by a factor of M - 1, and creates runs longer by the same factor.
    - E.g. If M=11, and there are 90 runs, one pass reduces the number of runs to 9, each 10 times the size of the initial runs
  - Repeated passes are performed till all runs have been merged into one.
  - Total cost = $b_r(2\lceil \log_{M-1}(b_r/M) \rceil + 1)$
    - where b_r is the file size in pages