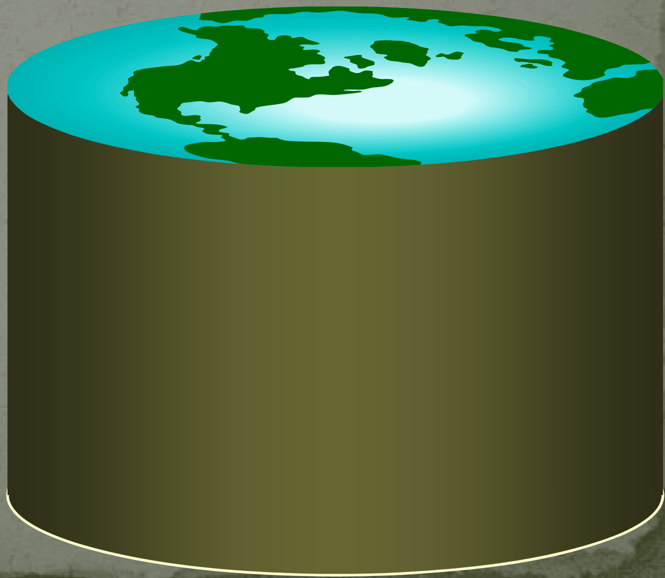
CS 505: Intermediate Topics to Database Systems

Instructor: Jinze Liu

Fall 2008



Review

- The unit of disk read and write is
  - Block (or called Page)
- The disk access time is composed by
  - Seek time
  - Rotation time
  - Data transfer time

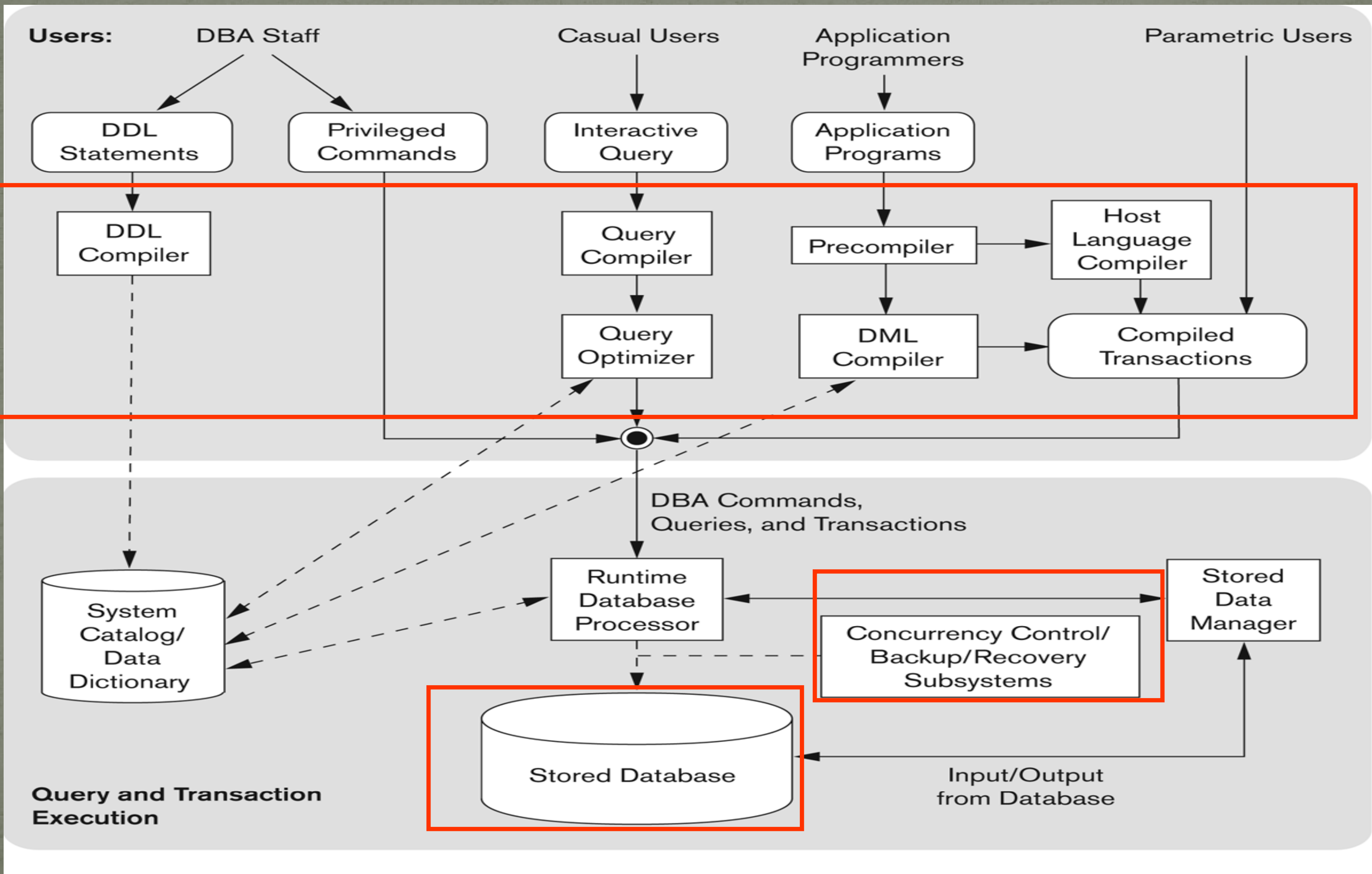
Review

- A row in a table, when located on disks, is called
  - A record
- Two types of record:
  - Fixed-length
  - Variable-length

Review

- In an abstract sense, a file is
  - A set of "records" on a disk
- In reality, a file is
  - A set of disk pages
- Each record lives on
  - A page
- Physical Record ID (RID)
  - A tuple of <page#, slot#>

A DBMS Preview



System Catalogs

- For each relation:
  - name, file location, file structure (e.g., Heap file)
  - attribute name and type, for each attribute
  - index name, for each index
  - integrity constraints
- For each index:
  - structure (e.g., B+ tree) and search key fields
- For each view:
  - view name and definition
- Plus statistics, authorization, buffer pool size, etc.

Catalogs are themselves stored as relations!

Attr_Cat(attr_name, rel_name, type, position)

attr_name	rel_name	type	position
attr_name	Attribute_Cat	string	1
rel_name	Attribute_Cat	string	2
type	Attribute_Cat	string	3
position	Attribute_Cat	integer	4
sid	Students	string	1
name	Students	string	2
login	Students	string	3
age	Students	integer	4
gpa	Students	real	5
fid	Faculty	string	1
fname	Faculty	string	2
sal	Faculty	real	3

Indexes

- A Heap file allows us to retrieve records:
  - by specifying the *rid*, or
  - by scanning all records sequentially
- Sometimes, we want to retrieve records by specifying the *values in one or more fields*, e.g.,
  - Find all students in the "CS" department
  - Find all students with a gpa > 3
- Indexes are file structures that enable us to answer such value-based queries efficiently.

Today's Topic

- How to locate data in a file *fast?*
- Introduction to indexing
- Tree-based indexes
  - ISAM: Indexed sequence access method
  - B^+-tree

Basics

- Given a value, locate the record(s) with this value

```
SELECT * FROM R WHERE A = value;
SELECT * FROM R, S WHERE R.A = S.B;
```

- Other search criteria, e.g.
  - Range search

```
SELECT * FROM R WHERE A > value;
```

  - Keyword search

database indexing   Search

Dense and sparse indexes

- Dense: one index entry for each search key value
- Sparse: one index entry for each block
  - Records must be clustered according to the search key

Dense versus sparse indexes

- Index size
  - Sparse index is smaller
- Requirement on records
  - Records must be clustered for sparse index
- Lookup
  - Sparse index is smaller and may fit in memory
  - Dense index can directly tell if a record exists
- Update
  - Easier for sparse index

Primary and secondary indexes

- Primary index
  - Created for the primary key of a table
  - Records are usually clustered according to the primary key
  - Can be sparse
- Secondary index
  - Usually dense
- SQL
  - `PRIMARY KEY` declaration automatically creates a primary index, `UNIQUE` key automatically creates a secondary index
  - Additional secondary index can be created on non-key attribute(s)

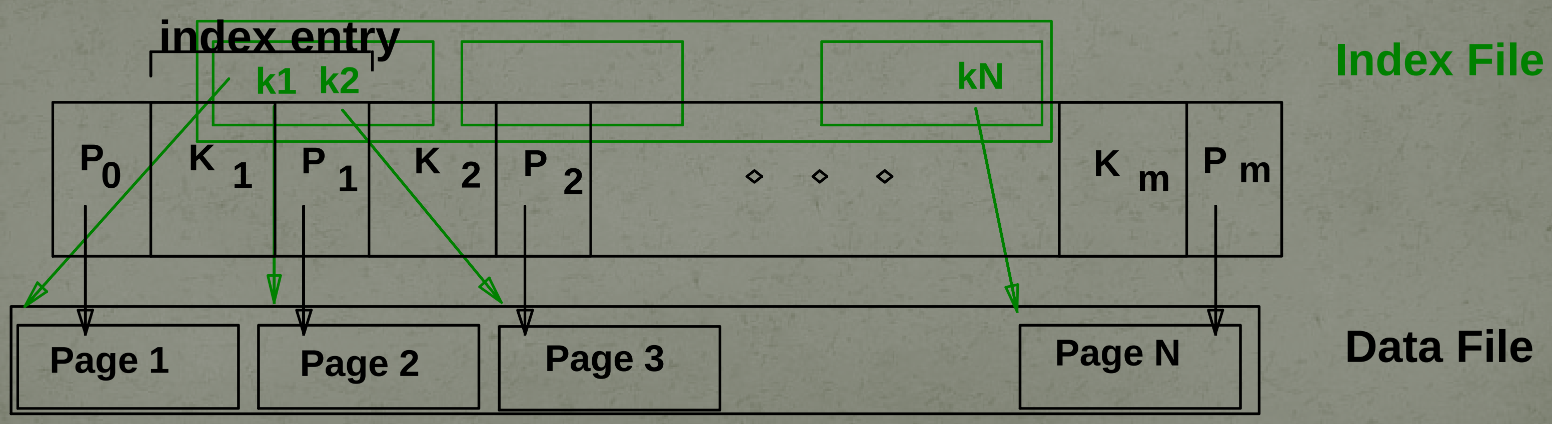
```
CREATE INDEX StudentGPAIndex ON
```

Tree-Structured Indexes: Introduction

- Tree-structured indexing techniques support both *range selections* and *equality selections*.
- ISAM =Indexed Sequential Access Method
  - static structure; early index technology.
- *B+ tree*: dynamic, adjusts gracefully under inserts and deletes.

Motivation for Index

- ``*Find all students with gpa > 3.0*"
  - If data file is sorted, do binary search
  - Cost of binary search in a database can be quite high, Why?
- Simple idea: Create an `index' file.

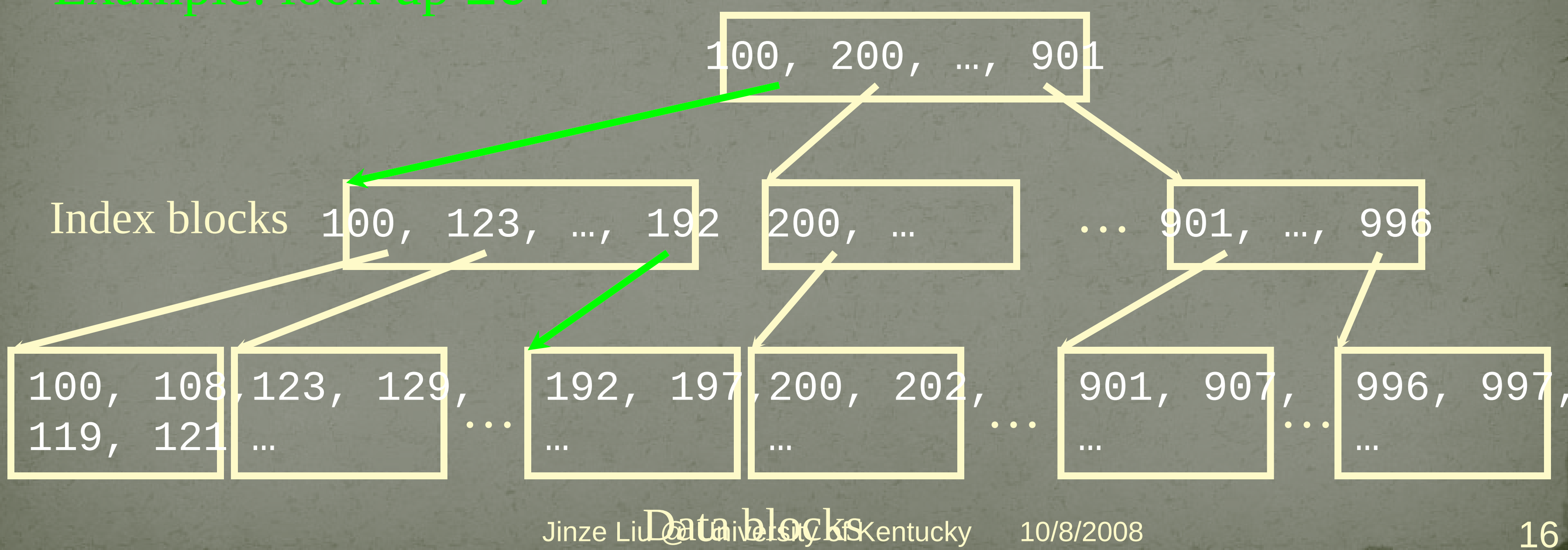


Can do binary search on (smaller) index file!

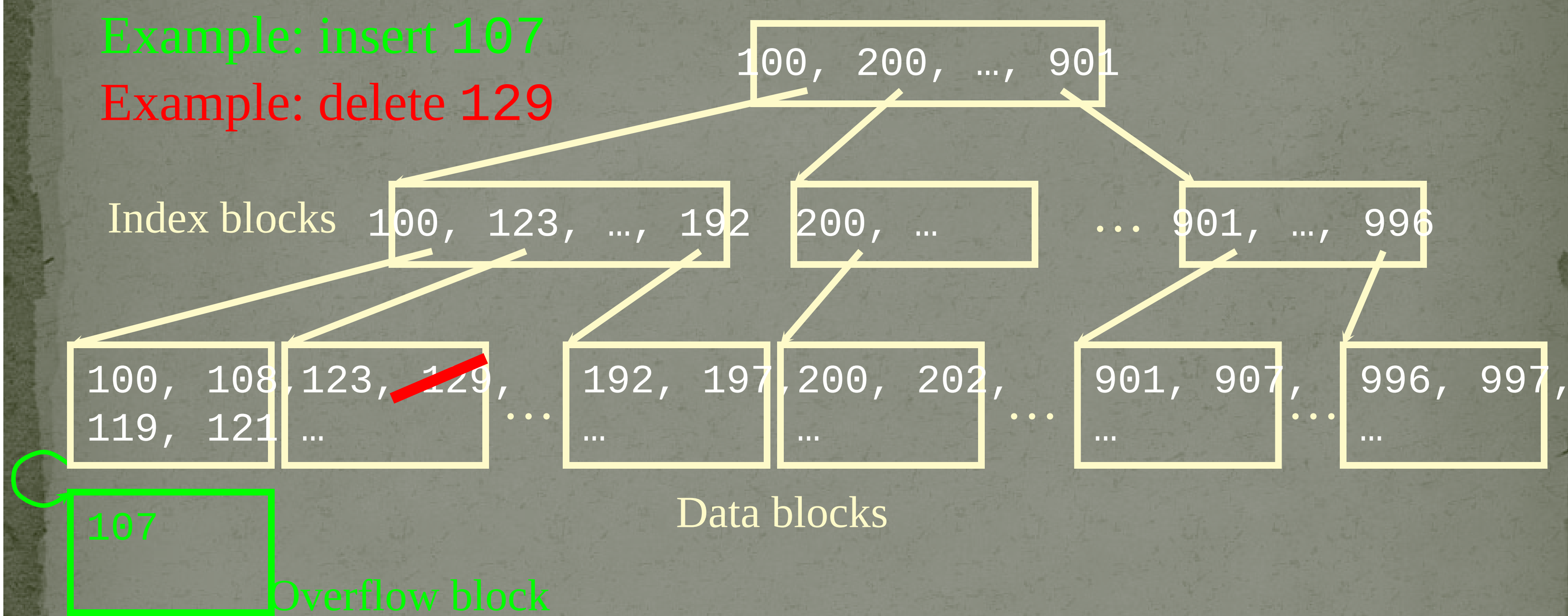
ISAM

- What if an index is still too big?
  - Put a another (sparse) index on top of that!
  - ☞ ISAM (Index Sequential Access Method), more or less

Example: look up 197



Updates with ISAM



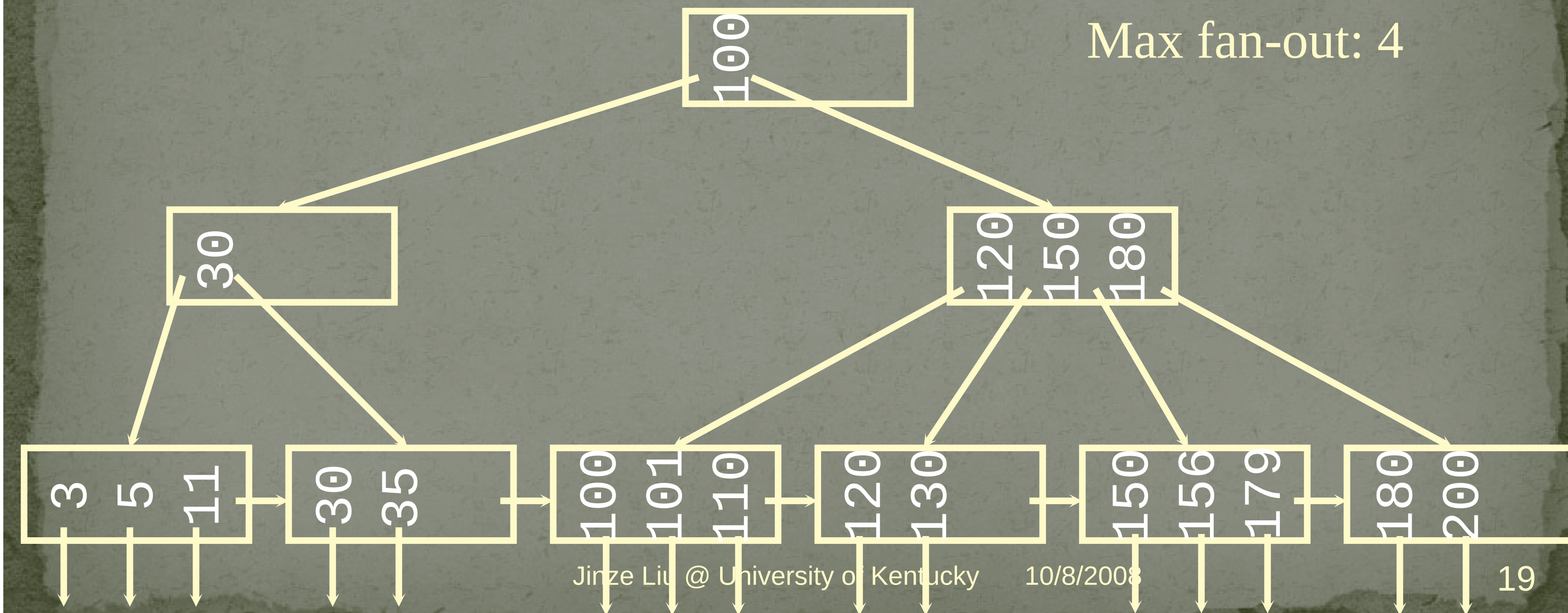
- Overflow chains and empty data blocks degrade performance
  - Worst case: most records go into one long chain

A Note of Caution

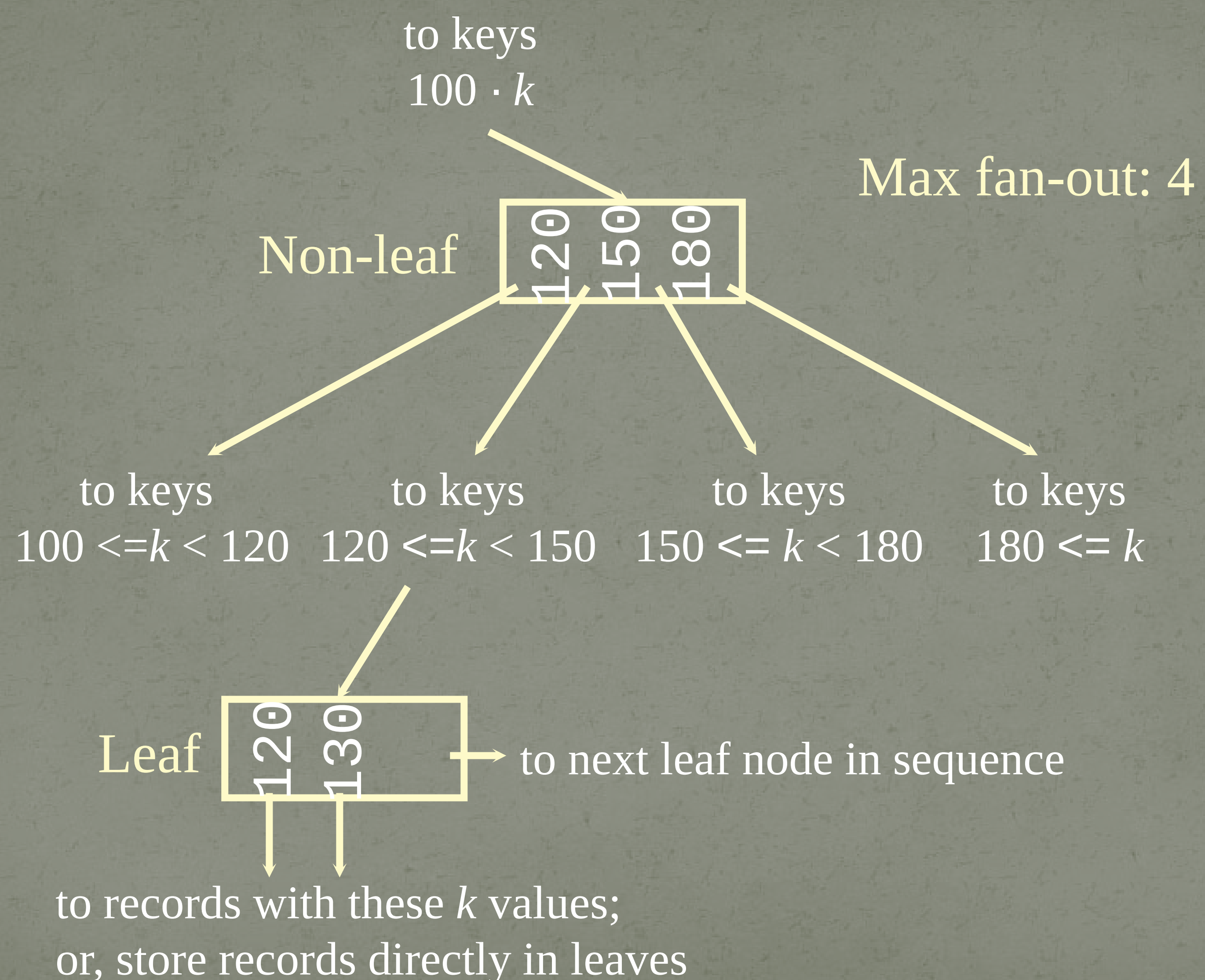
- ISAM is an old-fashioned idea
  - B+-trees are usually better, as we'll see
- But, ISAM is a good place to start to understand the idea of indexing
- Upshot
  - Don't brag about being an ISAM expert on your resume
  - Do understand how they work, and tradeoffs with B^+-trees

B+-tree

- A hierarchy of intervals
- Balanced (more or less): good performance guarantee
- Disk-based: one node per block; large fan-out



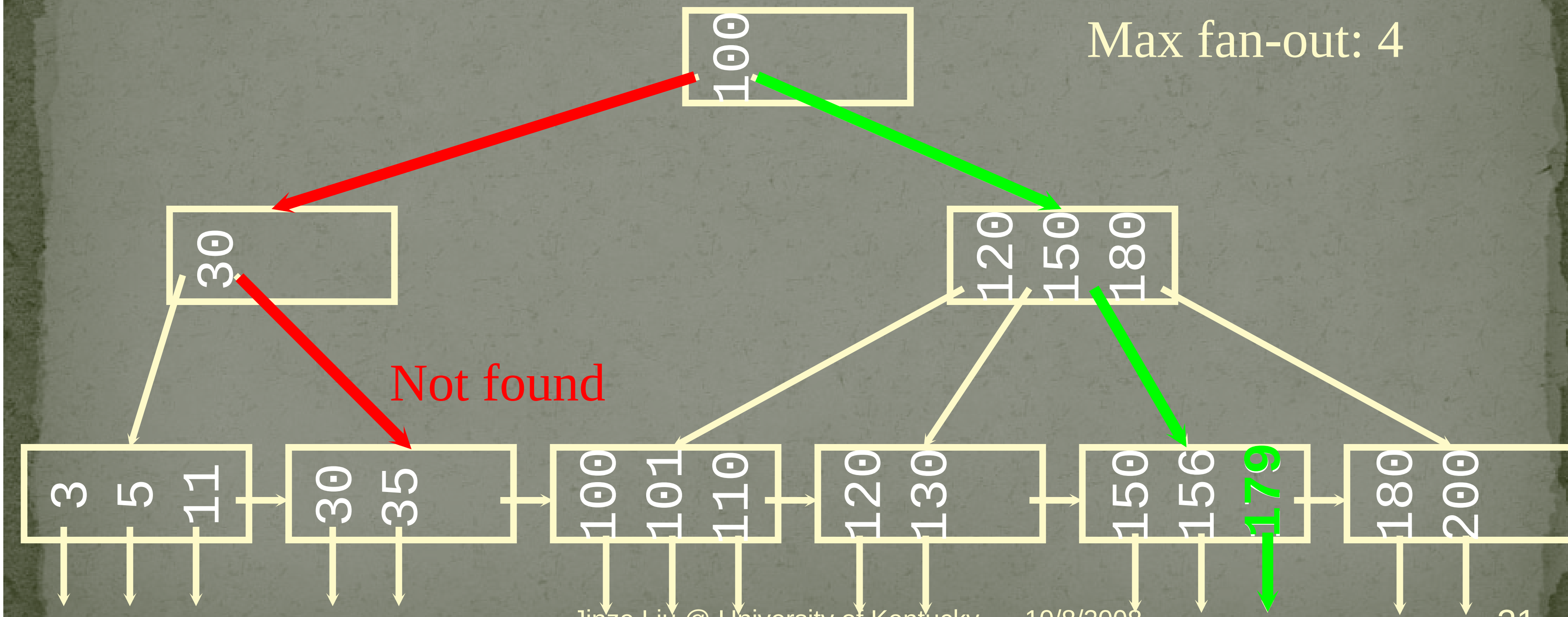
Sample B+-tree nodes



Lookups

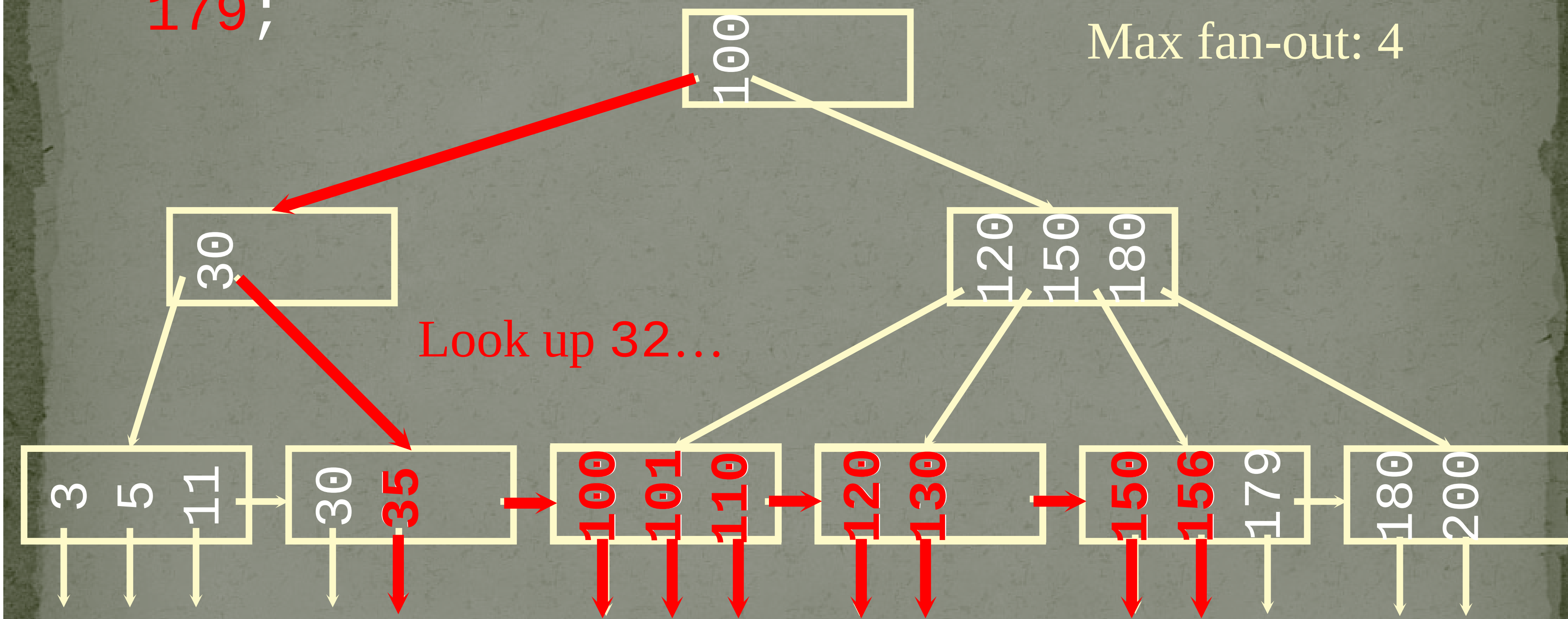
SELECT * FROM *R* WHERE *k* = 179;

SELECT * FROM *R* WHERE *k* = 32;



Range query

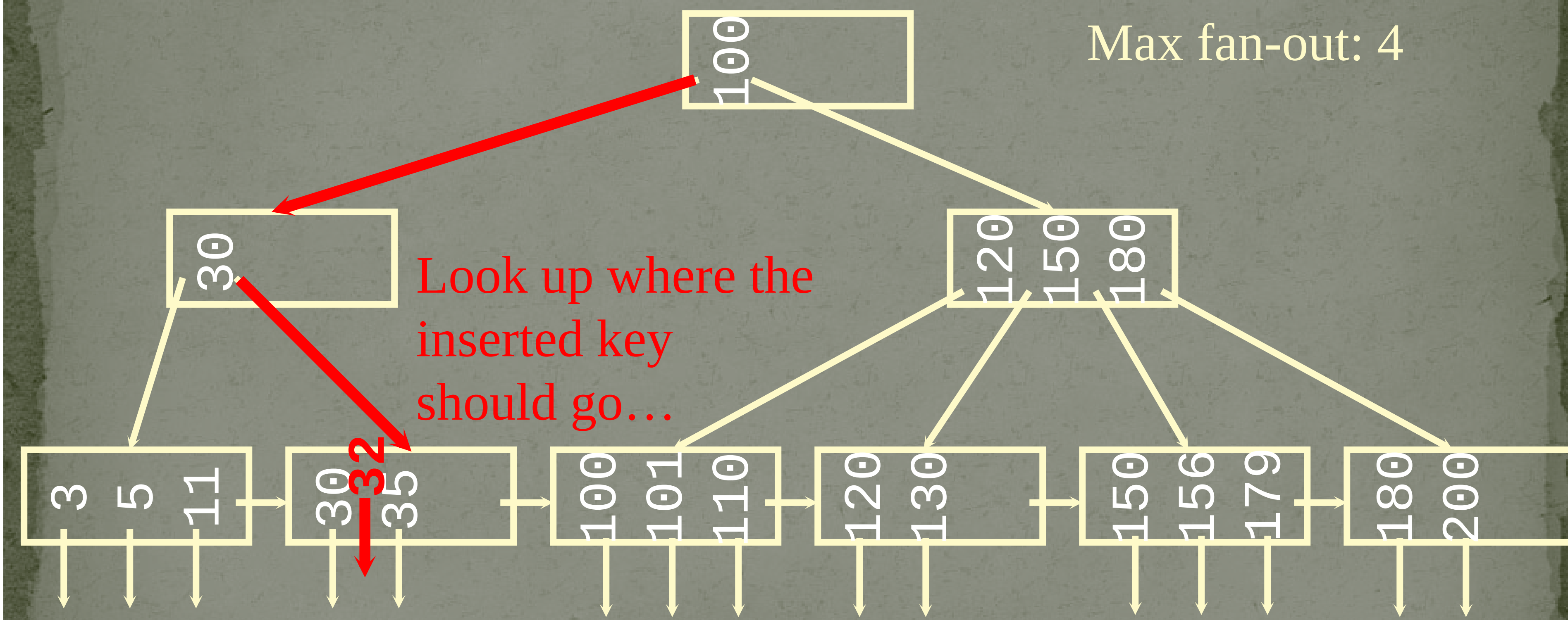
SELECT * FROM *R* WHERE *k* > 32 AND *k* <
179;



And follow next-leaf pointers

Insertion

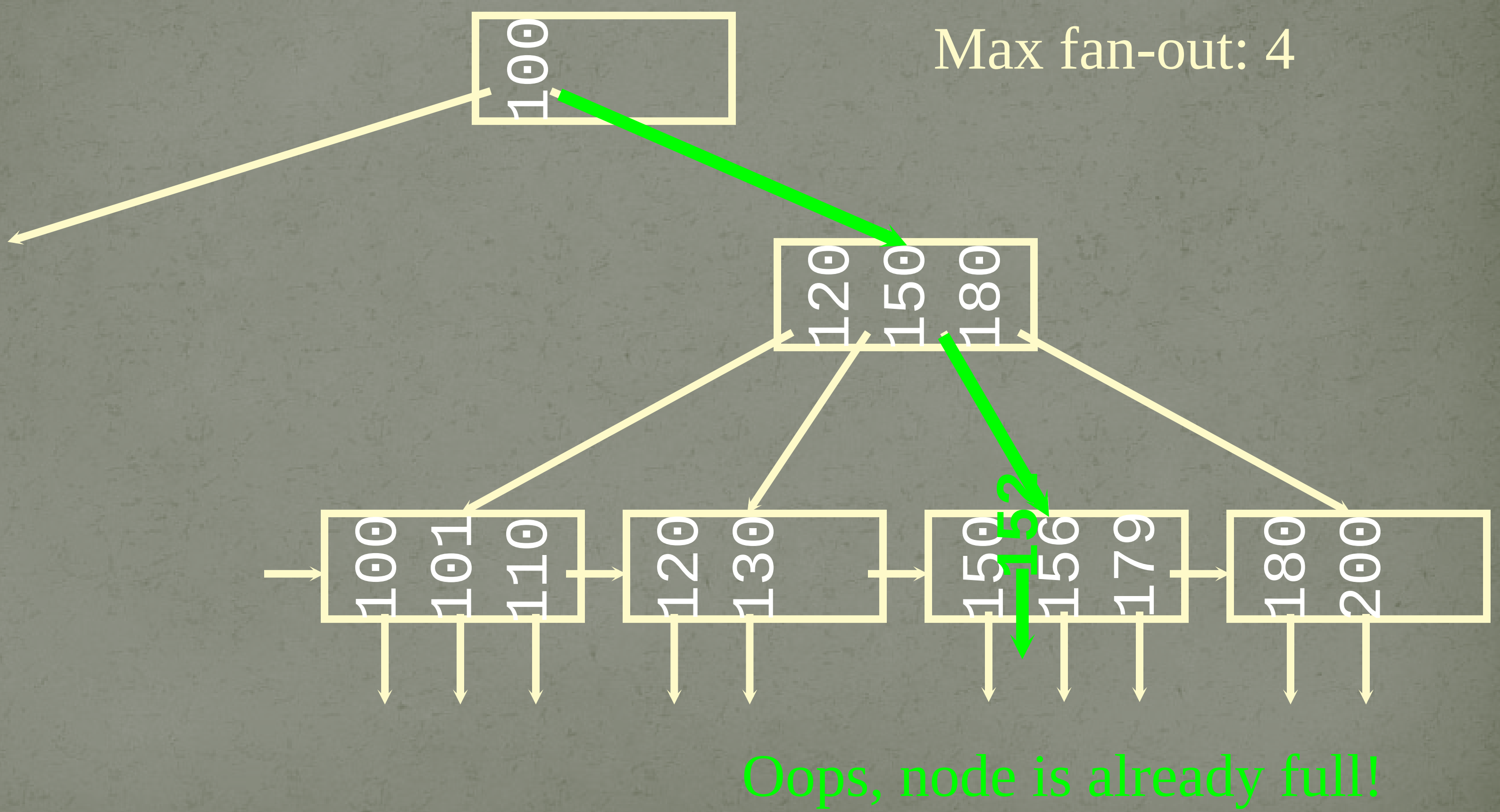
- Insert a record with search key value 32



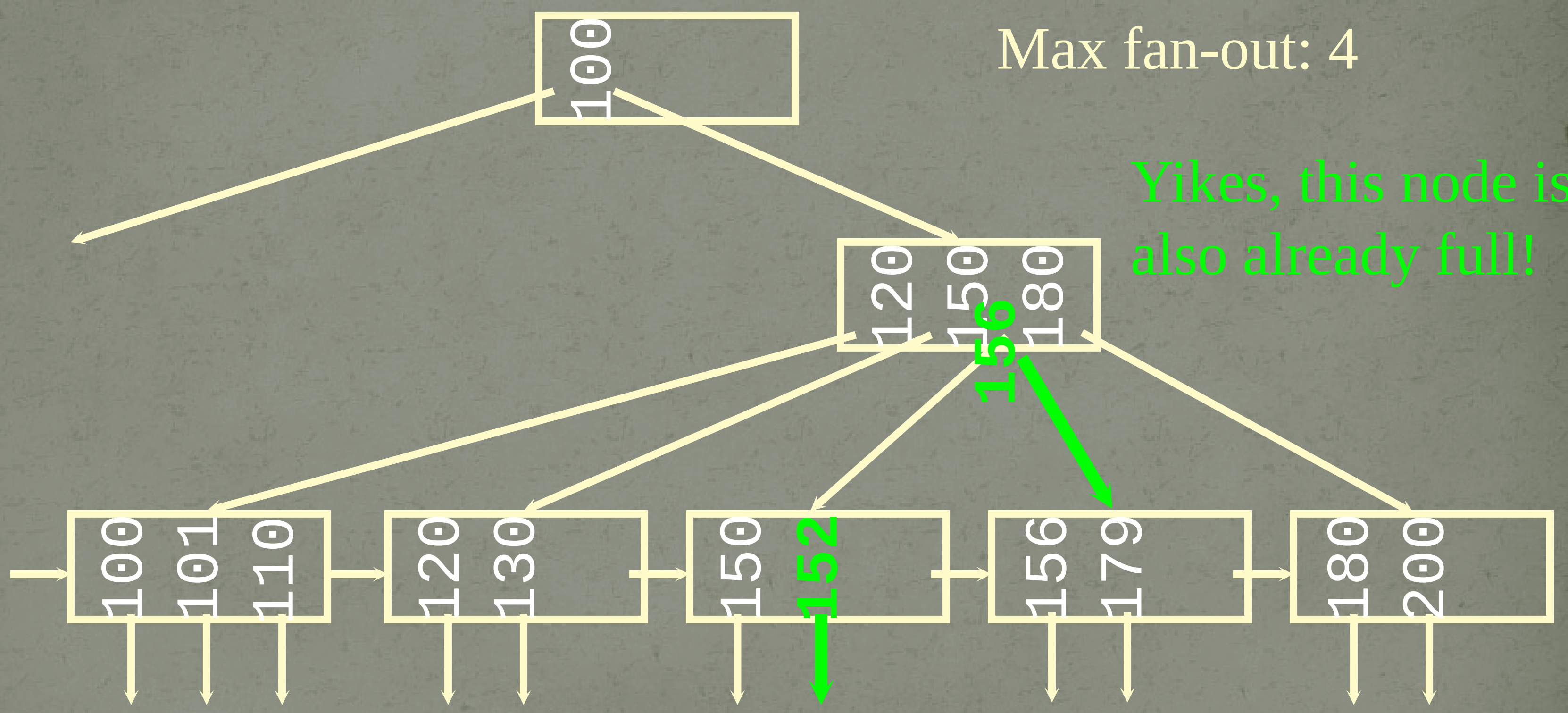
And insert it right there

Another insertion example

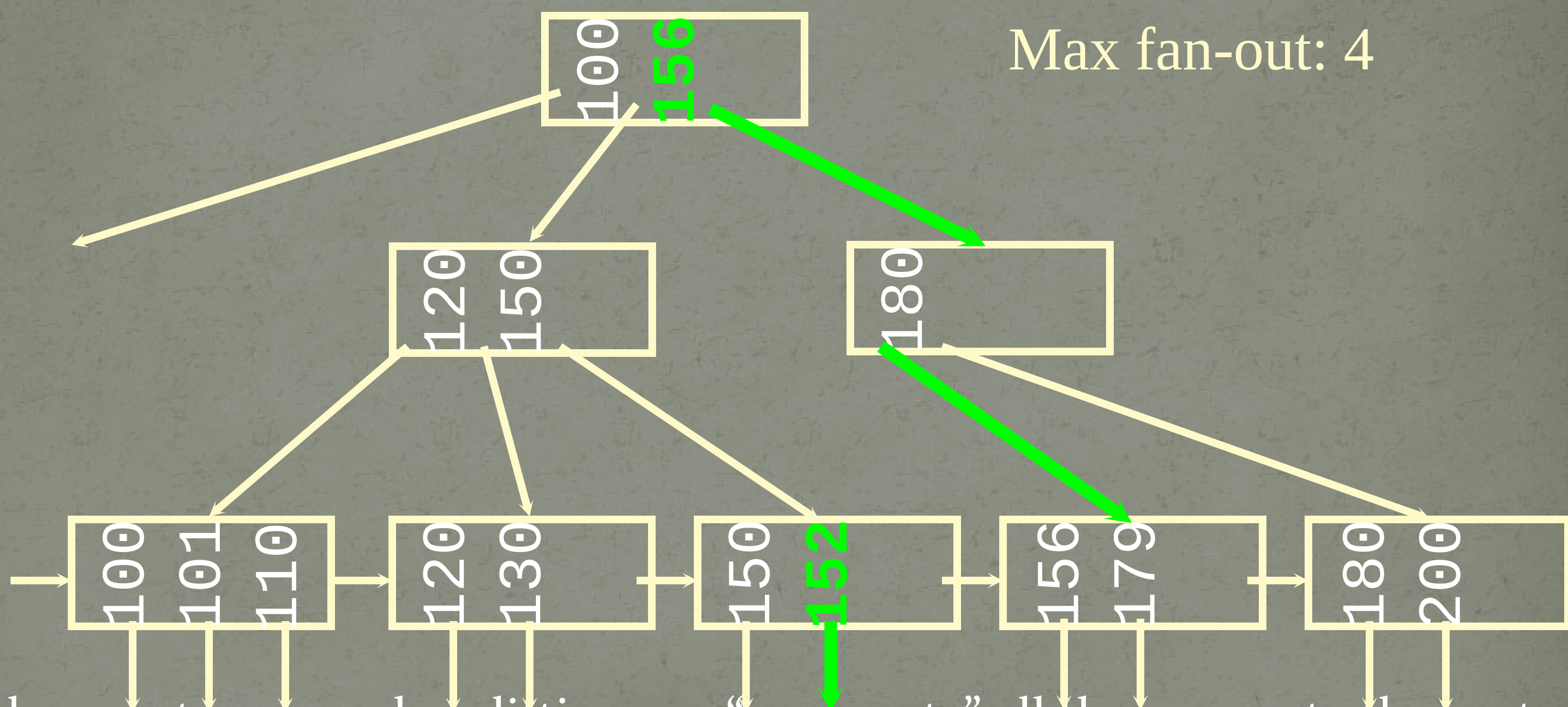
- Insert a record with search key value 152



Node splitting



More node splitting



Max fan-out: 4

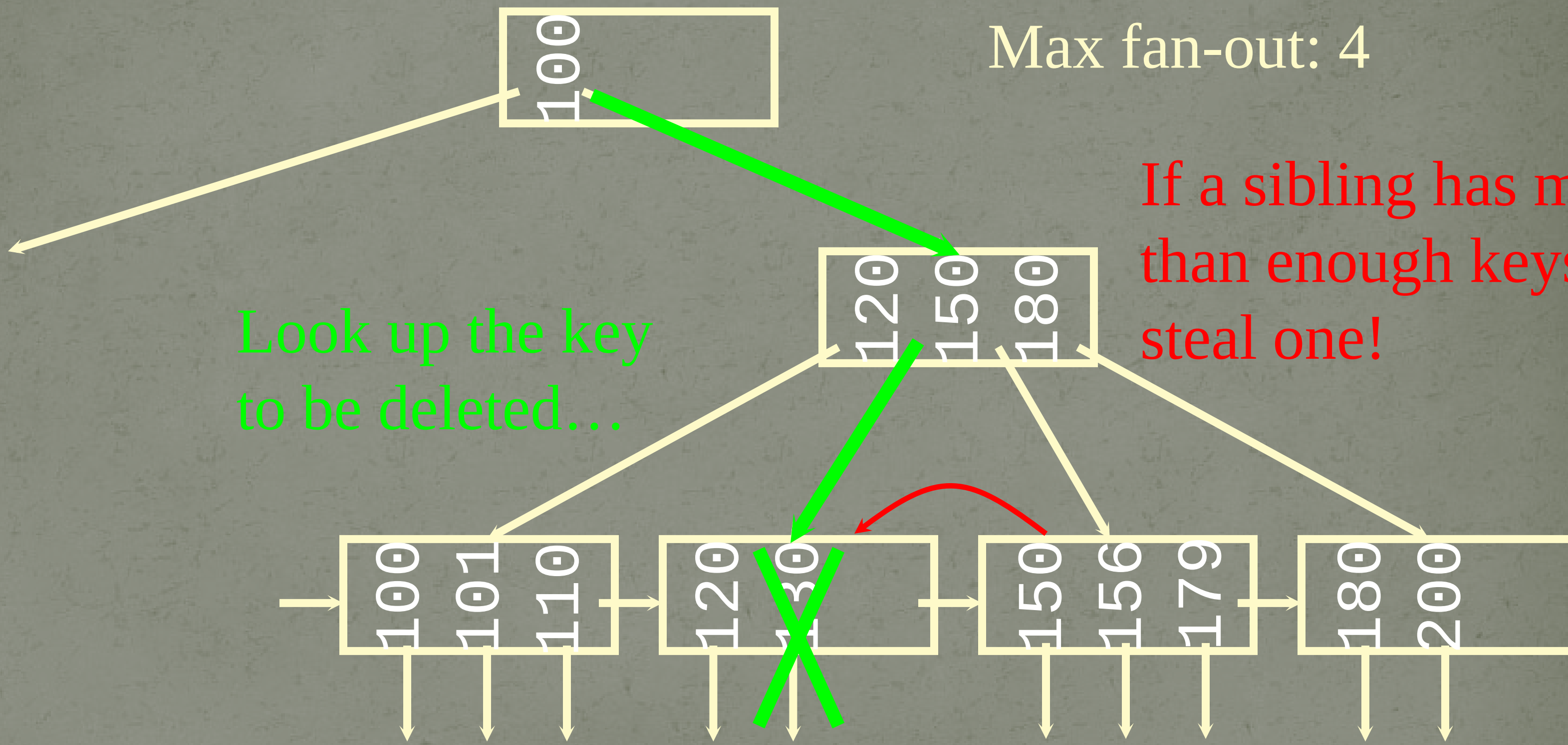
- In the worst case, node splitting can “propagate” all the way up to the root of the tree (not illustrated here)
  - Splitting the root introduces a new root of fan-out 2 and causes the tree to grow “up” by one level

Insertion

- B^+-tree Insert
- Find correct leaf *L*.
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Distribute entries evenly, *copy up* middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
- Tree growth: gets **wider** and (sometimes) **one level taller at top.**

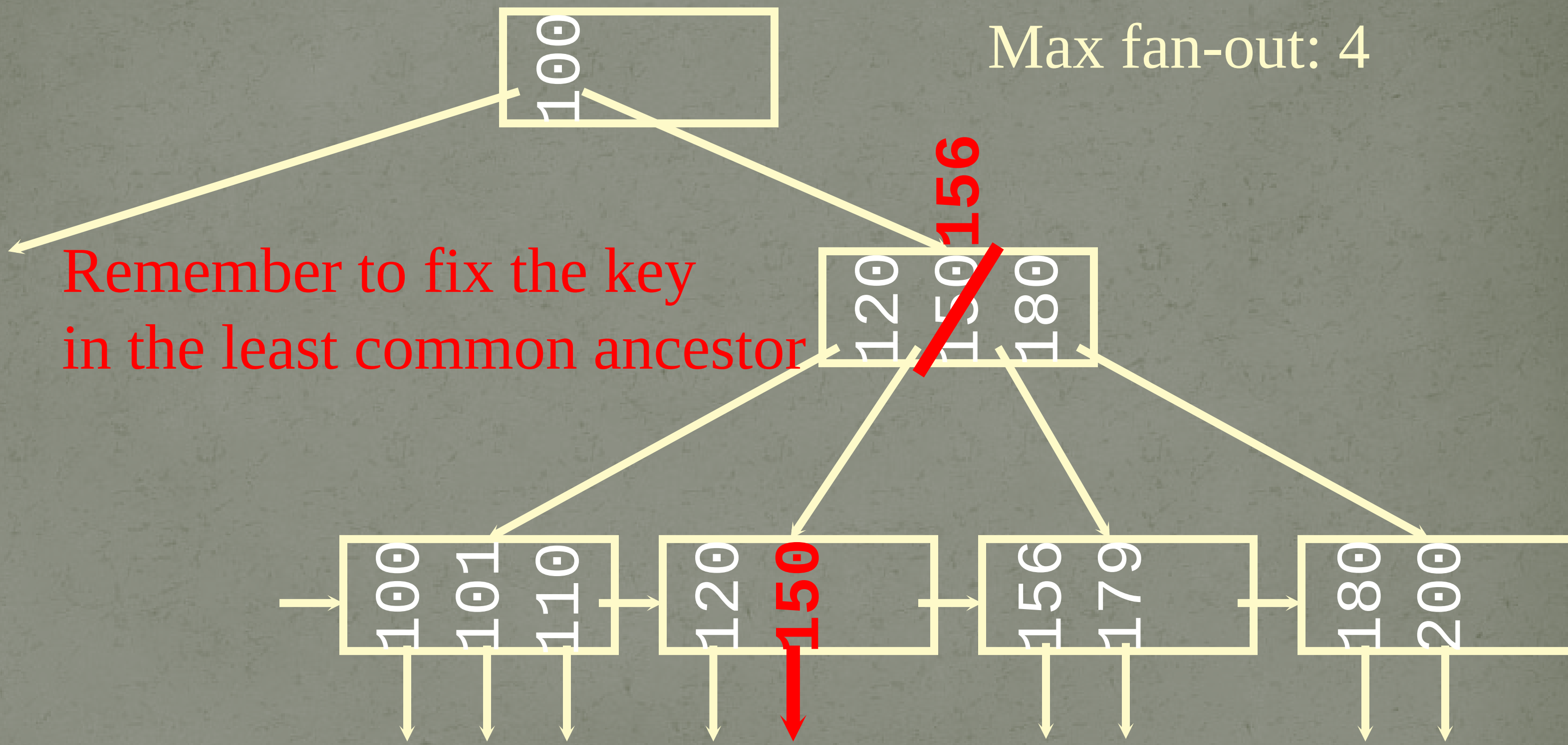
Deletion

- Delete a record with search key value `130`



Stealing from a sibling

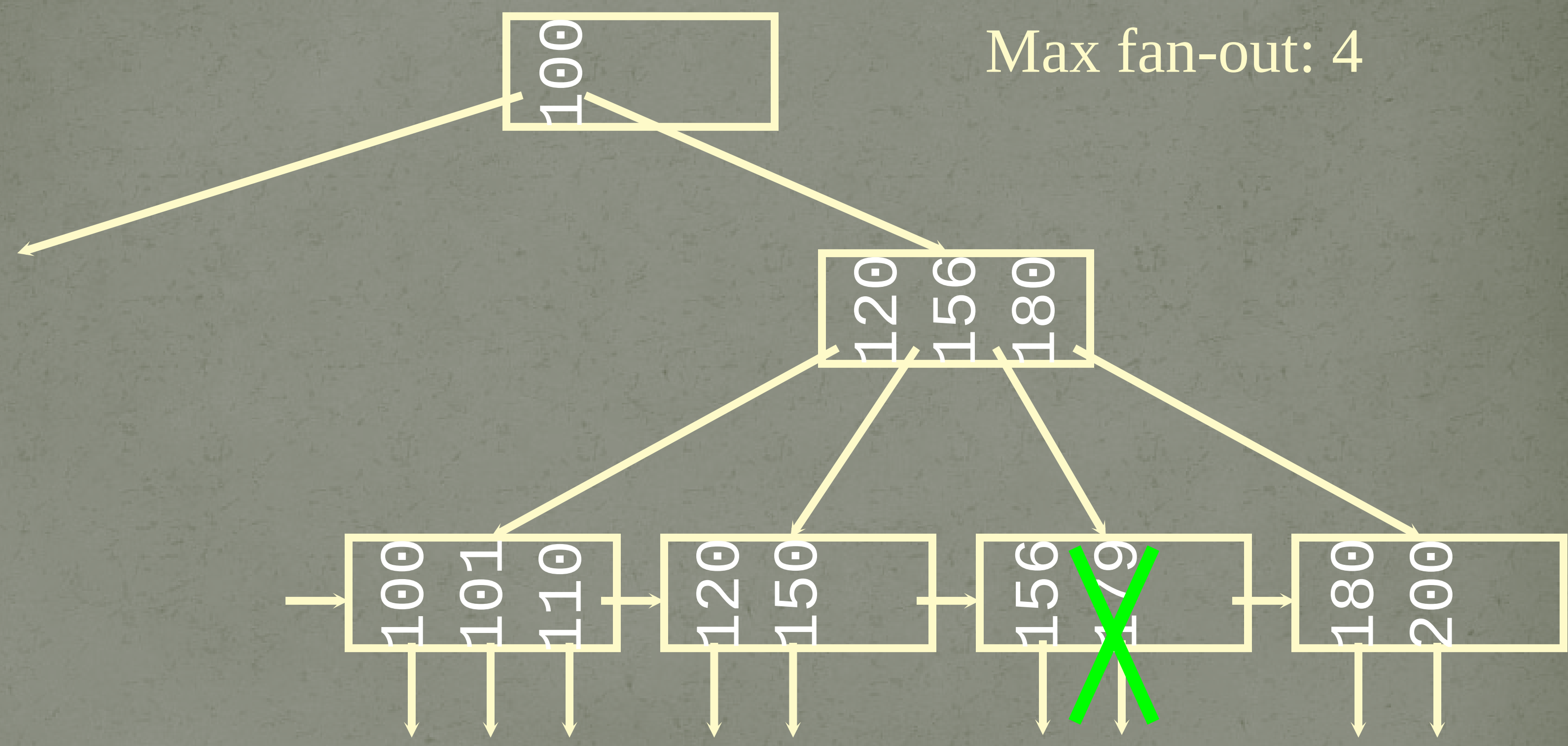
Max fan-out: 4



Remember to fix the key
in the least common ancestor

Another deletion example

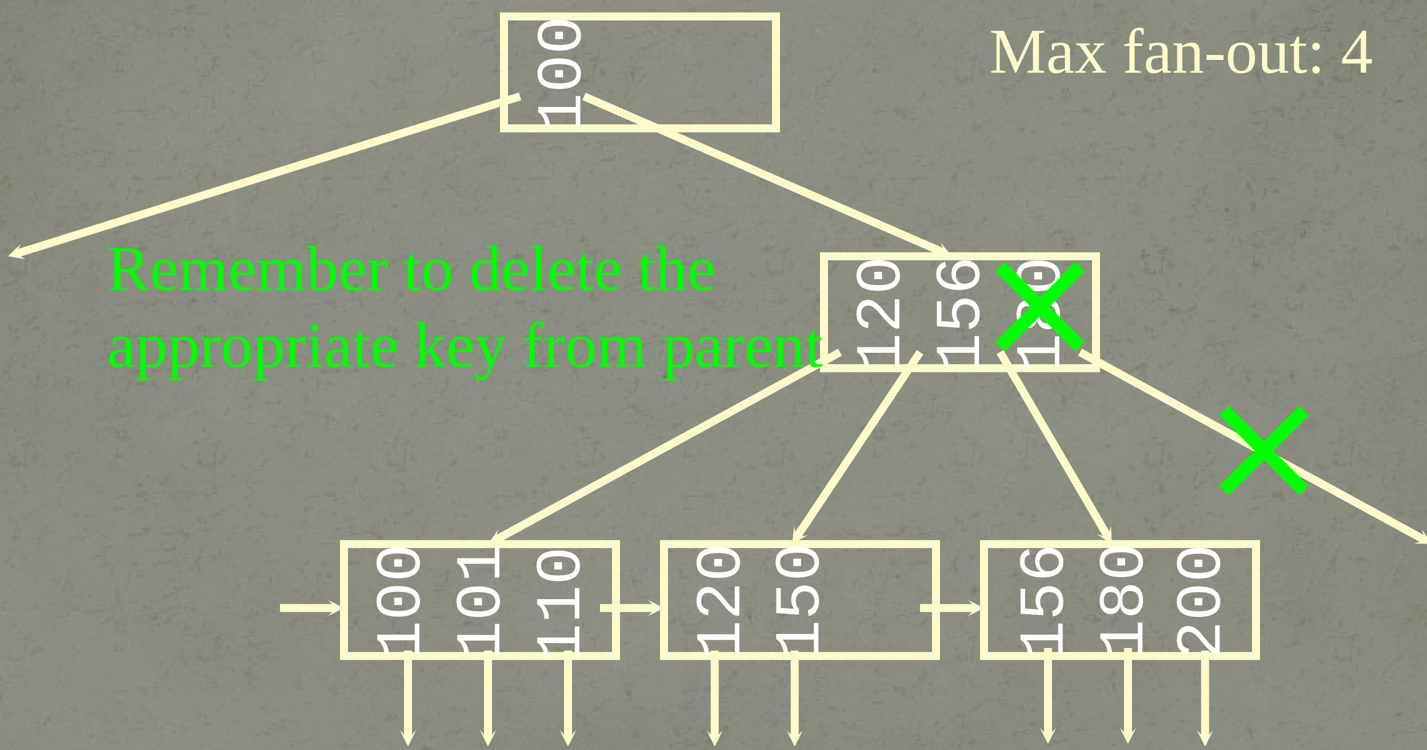
- Delete a record with search key value 179



Max fan-out: 4

Cannot steal from siblings
Then coalesce (merge) with a sibling!

Coalescing

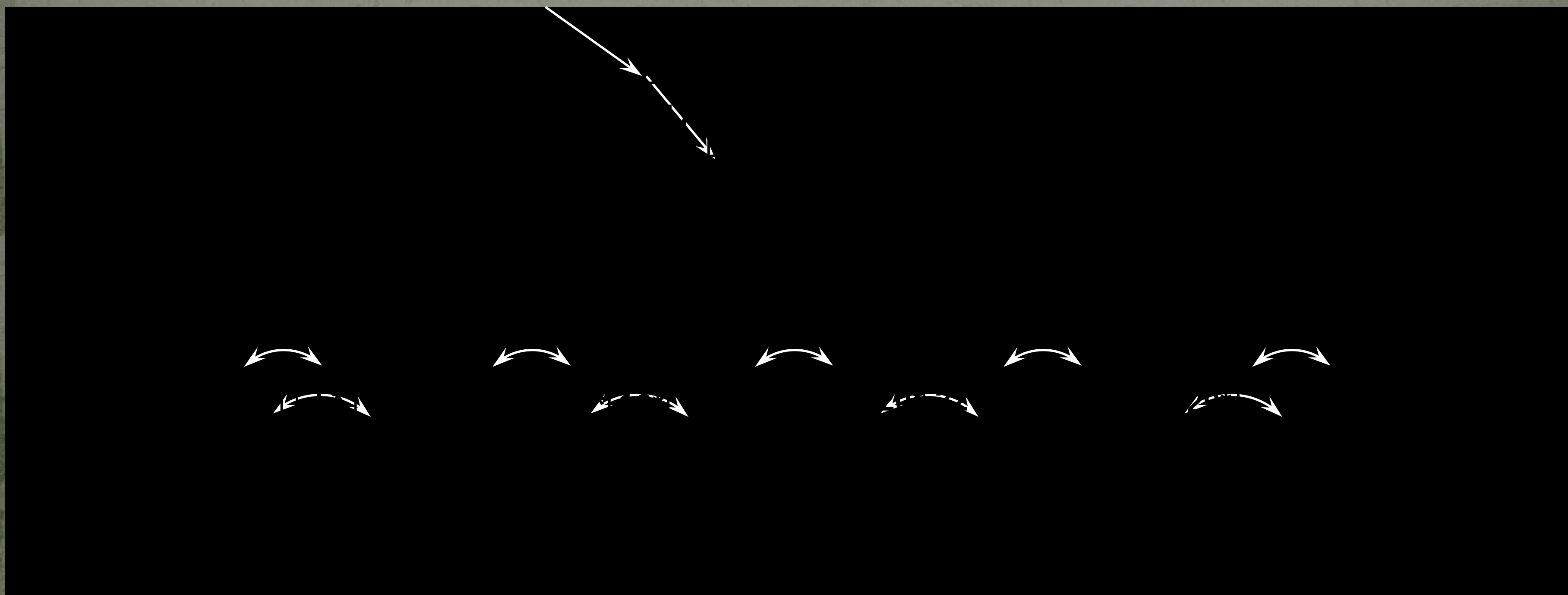


- Deletion can "propagate" all the way up to the root of the tree (not illustrated here)
  - When the root becomes empty, the tree "shrinks" by one level

Deletion

- B+-tree Delete
- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to *redistribute*, borrowing from sibling *(adjacent node with same parent as L)*.
    - If re-distribution fails, *merge* *L* and sibling.
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- Tree shrink: gets **narrower** and (sometimes) **one level lower at top.**

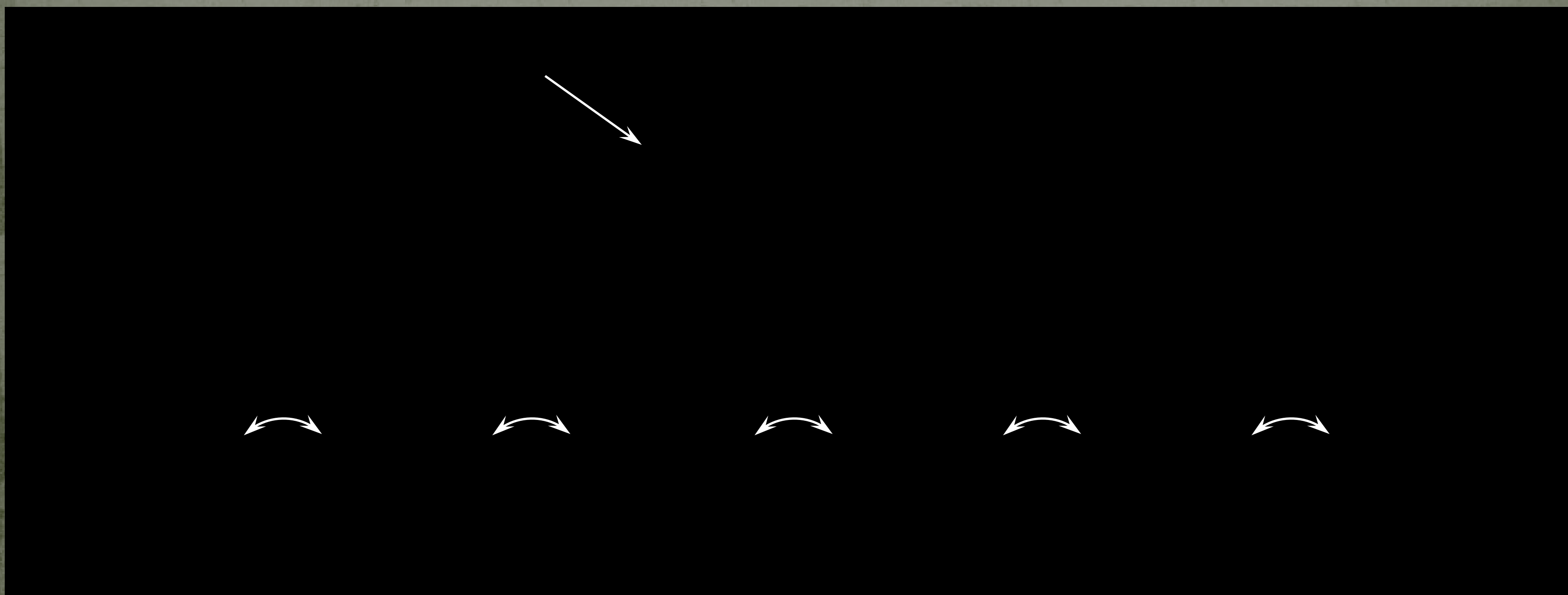
Example B+ Tree - Inserting 8*



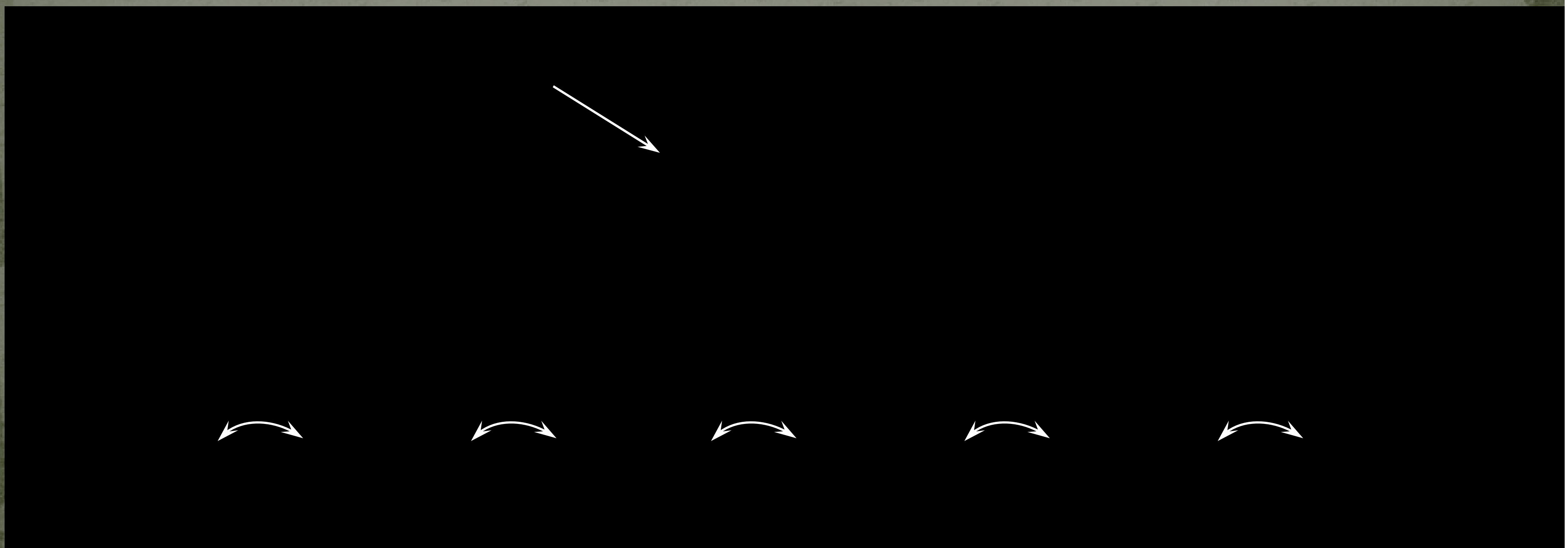
Notice that root was split, leading to increase in height.

In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

Example Tree (including 8*) Delete 19* and 20* ...



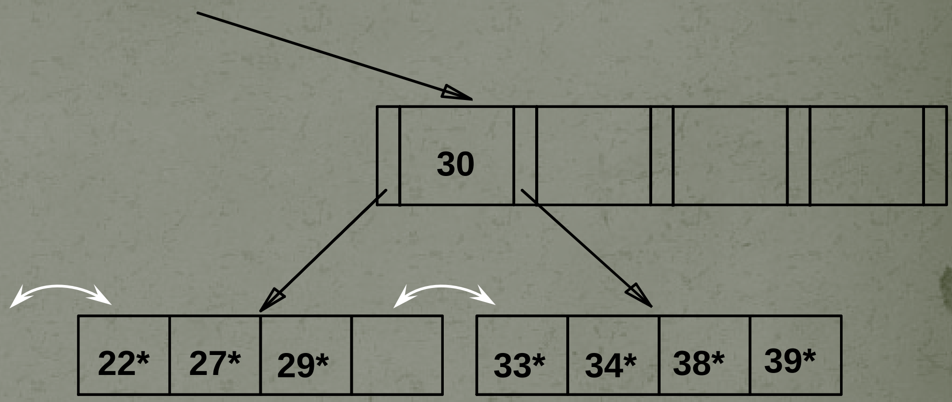
Example Tree (including 8*) Delete 19* and 20* ...

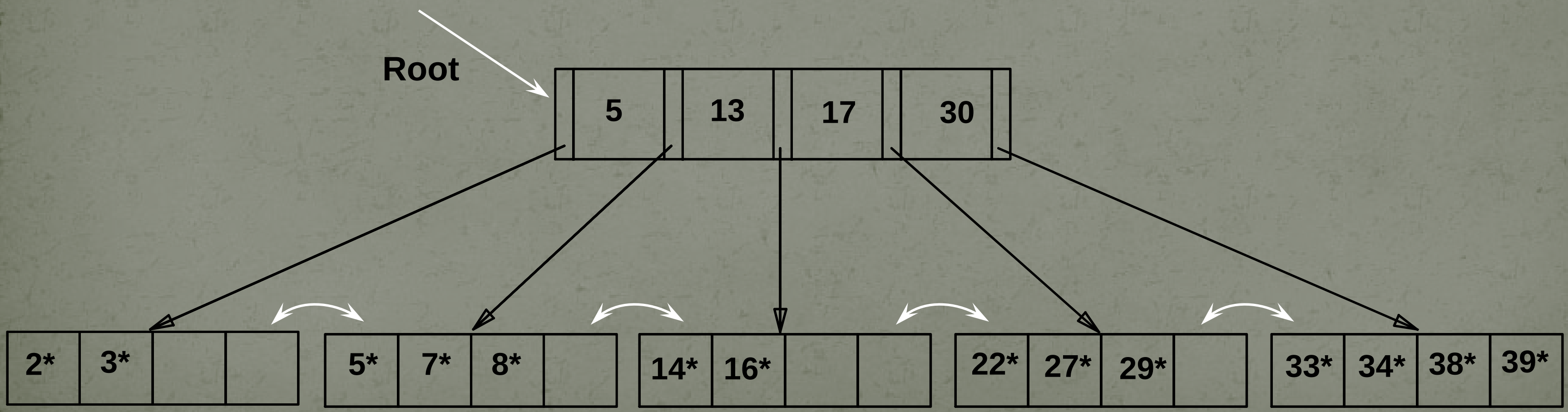


- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

... And Then Deleting 24*

- Must merge.
- Observe *`toss'* of index entry (key 27 on right), and *`pull down'* of index entry (below).





B^+-tree balancing properties

- Height constraint: all leaves at the same lowest level
- Fan-out constraint: all nodes at least half full (except root)

	Max # pointers	Max # keys	Min # active pointers	Min # keys
Non-leaf	f	$f-1$	$\lceil f/2 \rceil$	$\lceil f/2 \rceil - 1$
Root	f	$f-1$	2	1
Leaf	f	$f-1$	$\lfloor f/2 \rfloor$	$\lfloor f/2 \rfloor$

Performance analysis

- How many I/O's are required for each operation?
  - *h*, the height of the tree (more or less)
  - Plus one or two to manipulate actual records
  - Plus $O(h)$ for reorganization (should be very rare if f is large)
  - Minus one if we cache the root in memory
- How big is h?
  - Roughly $\log_{\text{fan-out}} N$, where N is the number of records
  - B^+-tree properties guarantee that fan-out is least f / 2 for all non-root nodes
  - Fan-out is typically large (in hundreds)—many keys and pointers can fit into one block
  - A 4-level B^+-tree is enough for typical tables

B^+-tree in practice

- Complex reorganization for deletion often is not implemented (e.g., Oracle, Informix)
  - Leave nodes less than half full and periodically reorganize
- Most commercial DBMS use B^+-tree instead of hashing-based indexes because B^+-tree handles range queries

The Halloween Problem

- Story from the early days of System R…

```
UPDATE Payroll
SET salary = salary * 1.1
WHERE salary >= 100000;
```

  - There is a B^+-tree index on *Payroll*(*salary*)
  - The update never stopped (why?)
- Solutions?
  - Scan index in reverse
  - Before update, scan index to create a complete "to-do" list
  - During update, maintain a "done" list
  - Tag every row with transaction/statement id

B^+-tree versus ISAM

- ISAM is more static; B^+-tree is more dynamic
- ISAM is more compact (at least initially)
  - Fewer levels and I/O's than B^+-tree
- Overtime, ISAM may not be balanced
  - Cannot provide guaranteed performance as B^+-tree does

B^+-tree versus B-tree

- B-tree: why not store records (or record pointers) in non-leaf nodes?
  - These records can be accessed with fewer I/O's
- Problems?
  - Storing more data in a node decreases fan-out and increases *h*
  - Records in leaves require more I/O's to access
  - Vast majority of the records live in leaves!

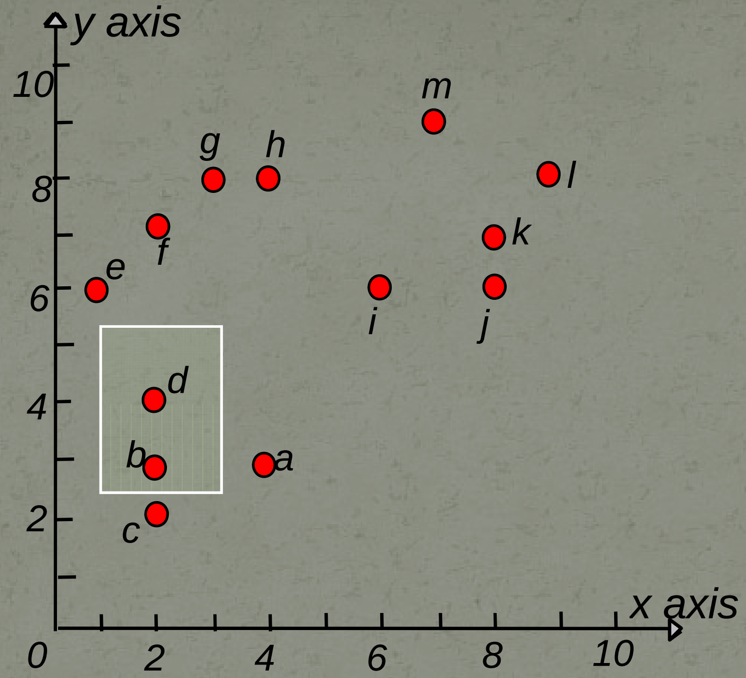
Beyond ISAM, B-, and B+-trees

- Other tree-based indexes: R-trees and variants, GiST, etc.
- Hashing-based indexes: extensible hashing, linear hashing, etc.
- Text indexes: inverted-list index, suffix arrays, etc.
- Other tricks: bitmap index, bit-sliced index, etc.
  - How about indexing subgraph search?

R-Tree

- The R-tree
  - Range Query
  - Aggregation Query
- NN Query
- RNN Query
- Closest Pair Query
- Close Pair Query
- Skyline Query

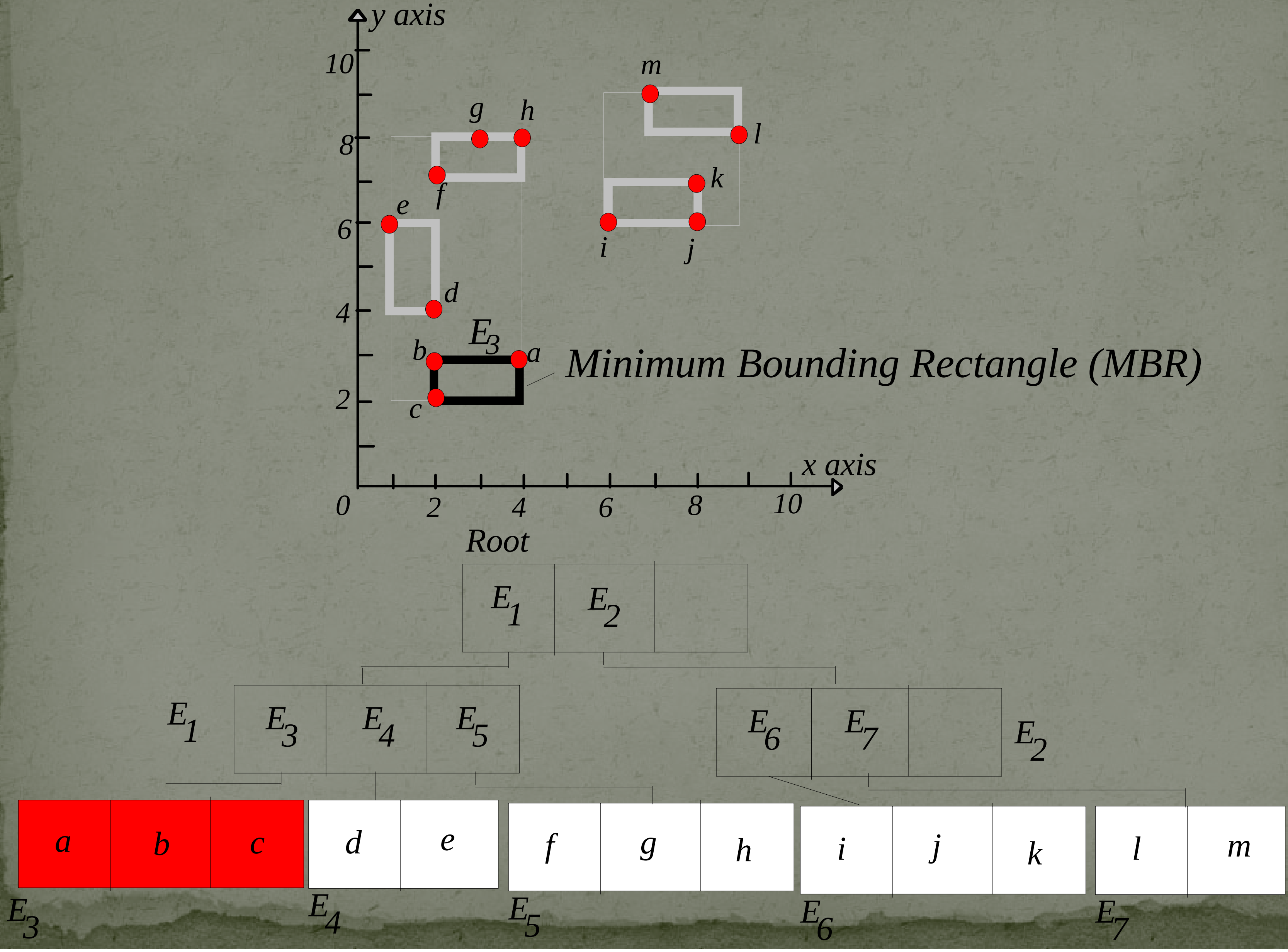
R-Tree Motivation



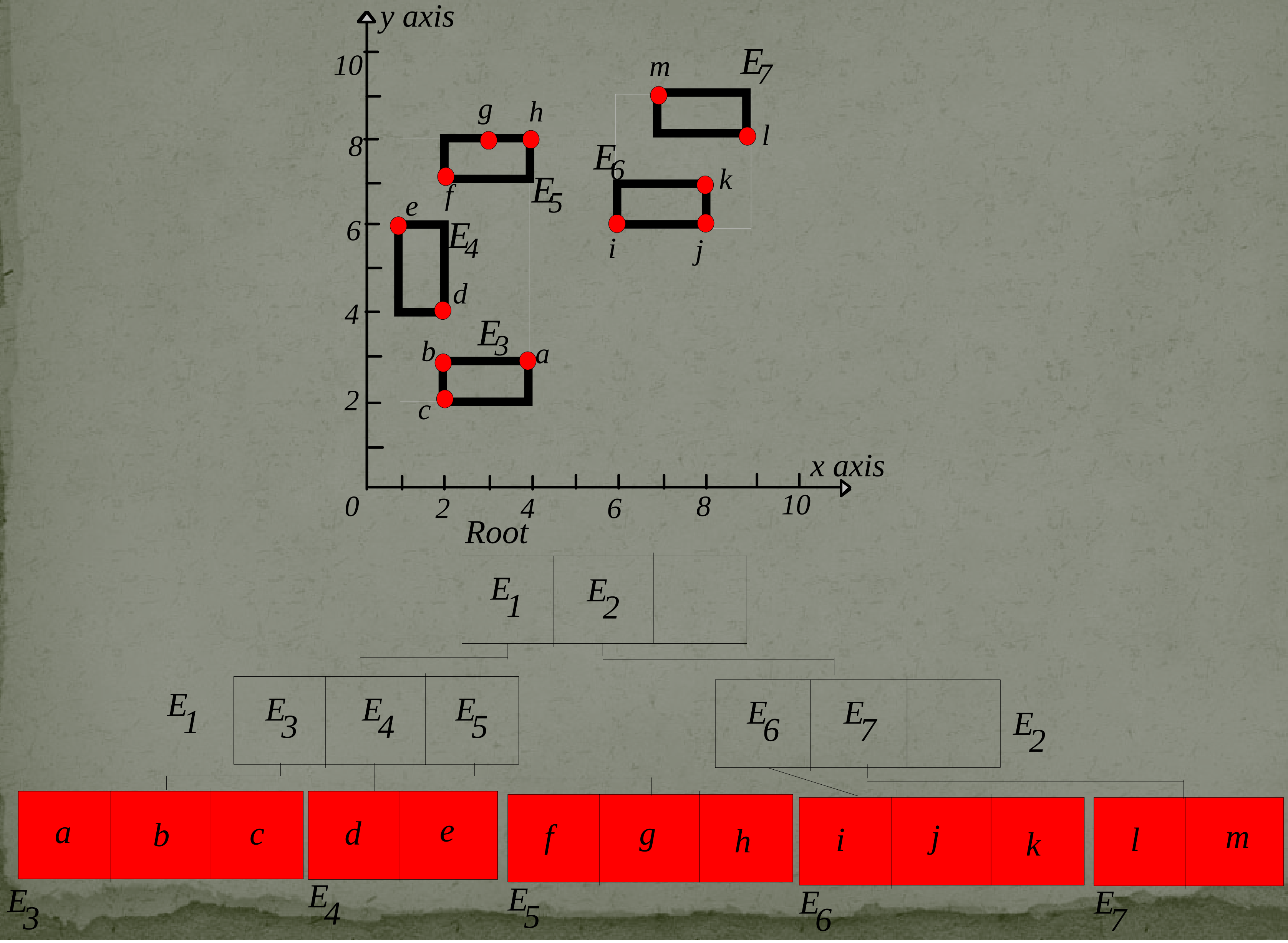
***Range query**: find the objects in a given range.*
E.g. find all hotels in Boston.

No index: scan through all objects. NOT EFFICIENT!

R-Tree: Clustering by Proximity

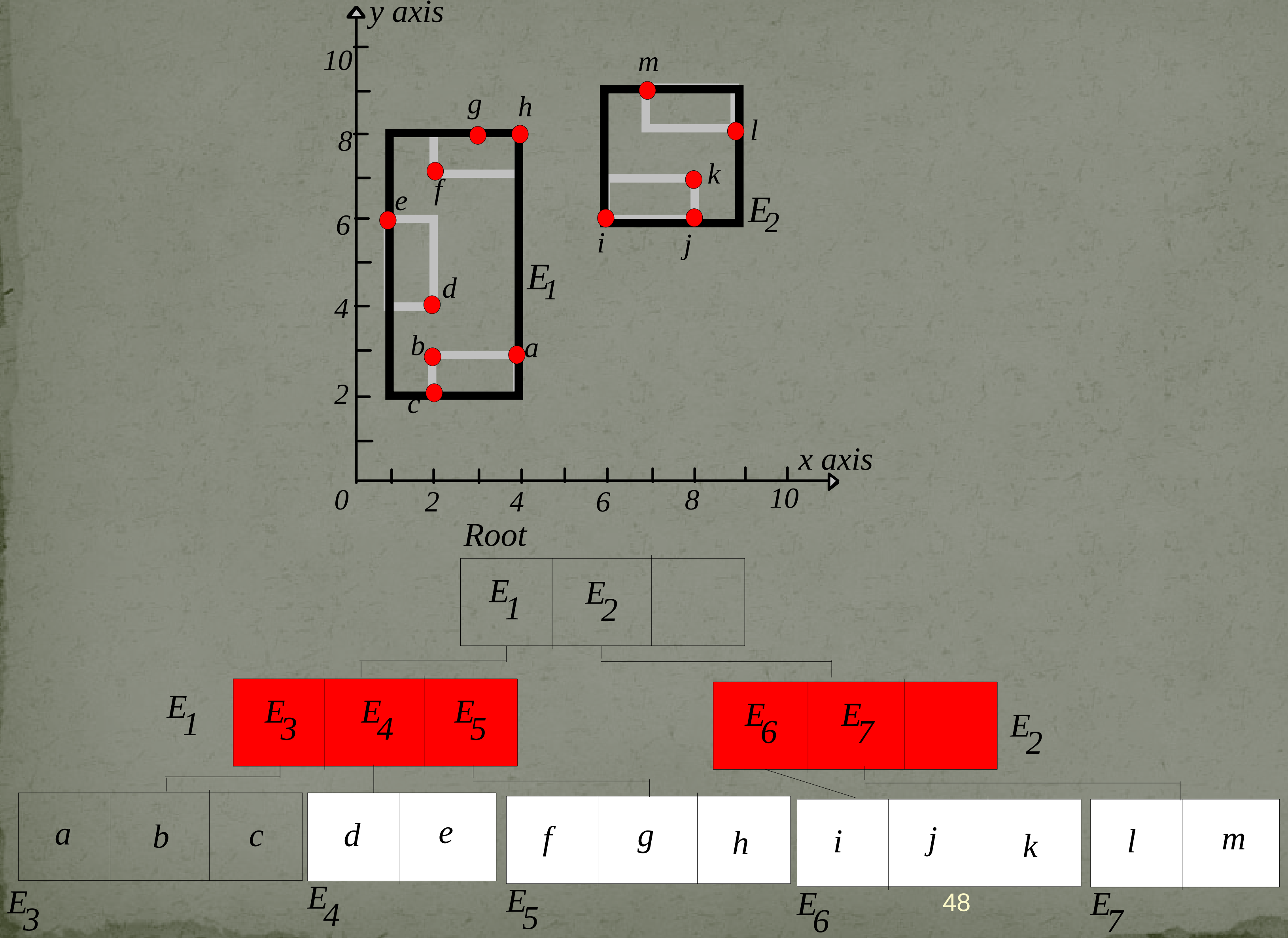


R-Tree



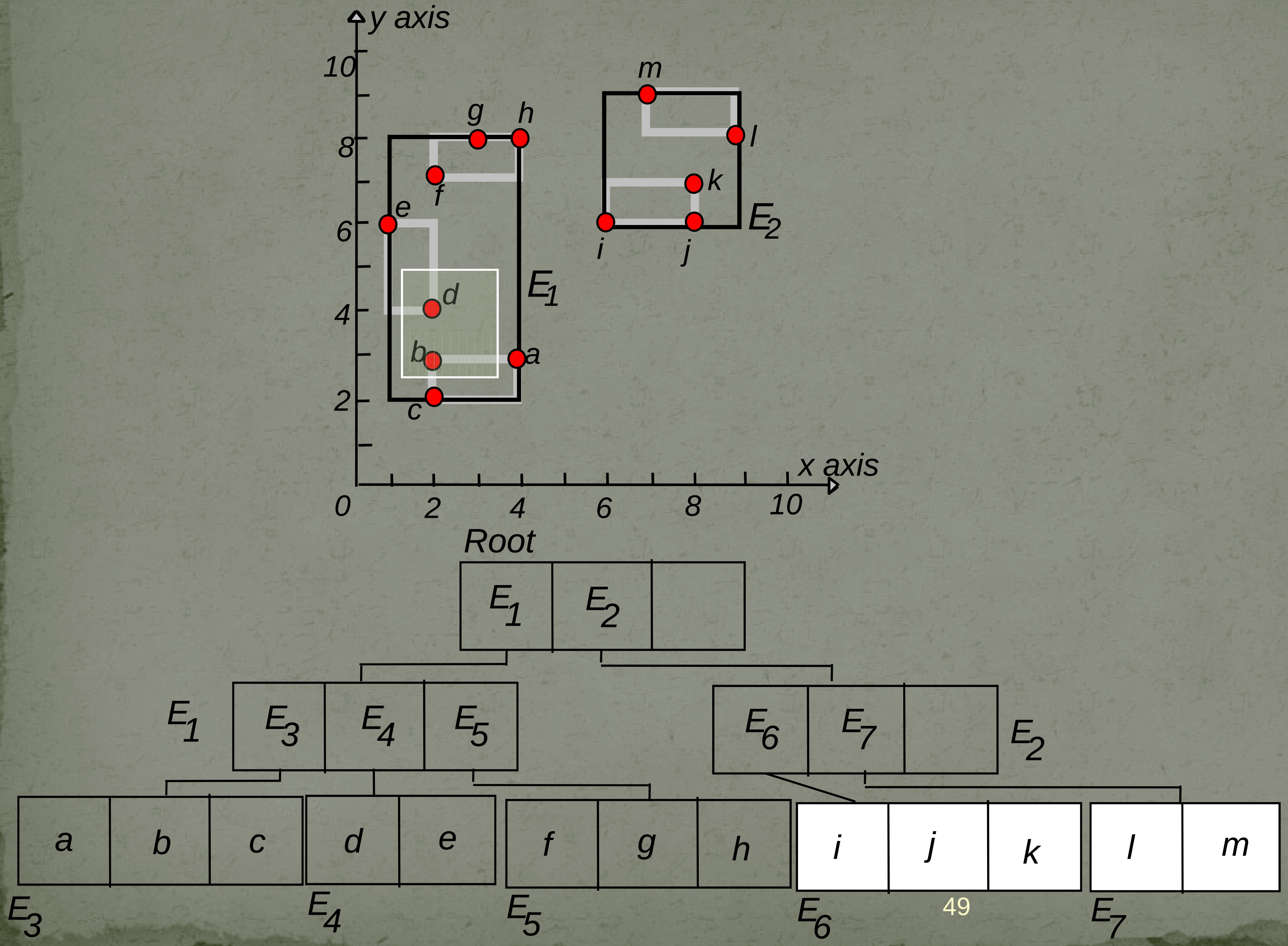
y axis
x axis
0
2
4
6
8
10
a
b
c
d
e
f
g
h
i
j
k
l
m
E_3
E_4
E_5
E_6
E_7
Root
E_1
E_2
E_1
E_3
E_4
E_5
E_6
E_7
E_2
a
b
c
d
e
f
g
h
i
j
k
l
m
E_3
E_4
E_5
E_6
E_7

R-Tree



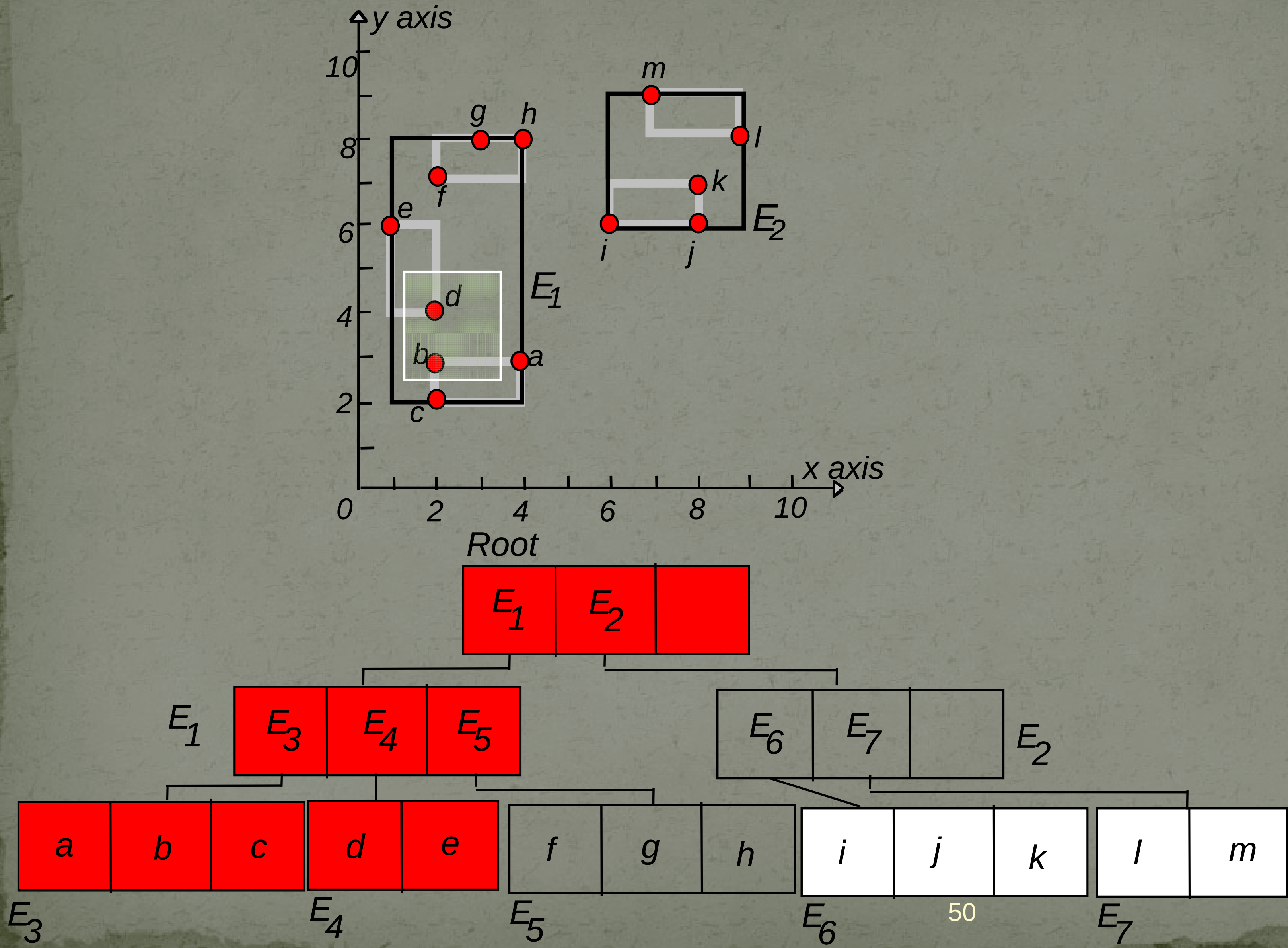
y axis
x axis
0
2
4
6
8
10
a
b
c
d
e
f
g
h
i
j
k
l
m
E1
E2
Root
E1
E2
E1
E3
E4
E5
E6
E7
E2
a
b
c
d
e
f
g
h
i
j
k
l
m
E3
E4
E5
E6
E7

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_3 E_4 E_5
E_6 E_7
a b c
d e
f g h
i j k
l m
E_3
E_4
E_5
E_6
E_7

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