

Data-intensive Computing Systems

Operators for Data Access

Shivnath Babu

Problem

- Relation: Employee (ID, Name, Dept, ...)
- 10 M tuples
- (Filter) Query:

```
SELECT *  
FROM Employee  
WHERE Name = "Bob"
```

Solution #1: Full Table Scan

- Storage:
 - Employee relation stored in **contiguous** blocks
- Query plan:
 - Scan the entire relation, output tuples with Name = "Bob"
- Cost:
 - Size of each record = 100 bytes
 - Size of relation = 10 M x 100 = 1 GB
 - Time @ 20 MB/s \approx 1 Minute

Solution #2

- Storage:
 - Employee relation **sorted** on Name attribute
- Query plan:
 - Binary search

Solution #2

- Cost:
 - Size of a block: 1024 bytes
 - Number of records per block: $1024 / 100 = 10$
 - Total number of blocks: $10 \text{ M} / 10 = 1 \text{ M}$
 - Blocks accessed by binary search: 20
 - Total time: $20 \text{ ms} \times 20 = 400 \text{ ms}$

Solution #2: Issues

- Filters on different attributes:

```
SELECT *  
FROM Employee  
WHERE Dept = "Sales"
```

- Inserts and Deletes

Indexes

- Data structures that efficiently evaluate a class of filter predicates over a relation
- Class of filter predicates:
 - Single or multi-attributes (**index-key attributes**)
 - Range and/or equality predicates
- (Usually) independent of physical storage of relation:
 - Multiple indexes per relation

Indexes

- Disk resident
 - Large to fit in memory
 - Persistent
- Updated when indexed relation updated
 - Relation updates costlier
 - Query cheaper

Problem

- Relation: Employee (ID, Name, Dept, ...)
- (Filter) Query:

```
SELECT *  
FROM Employee  
WHERE Name = "Bob"
```

Single-Attribute Index on **Name** that supports equality predicates

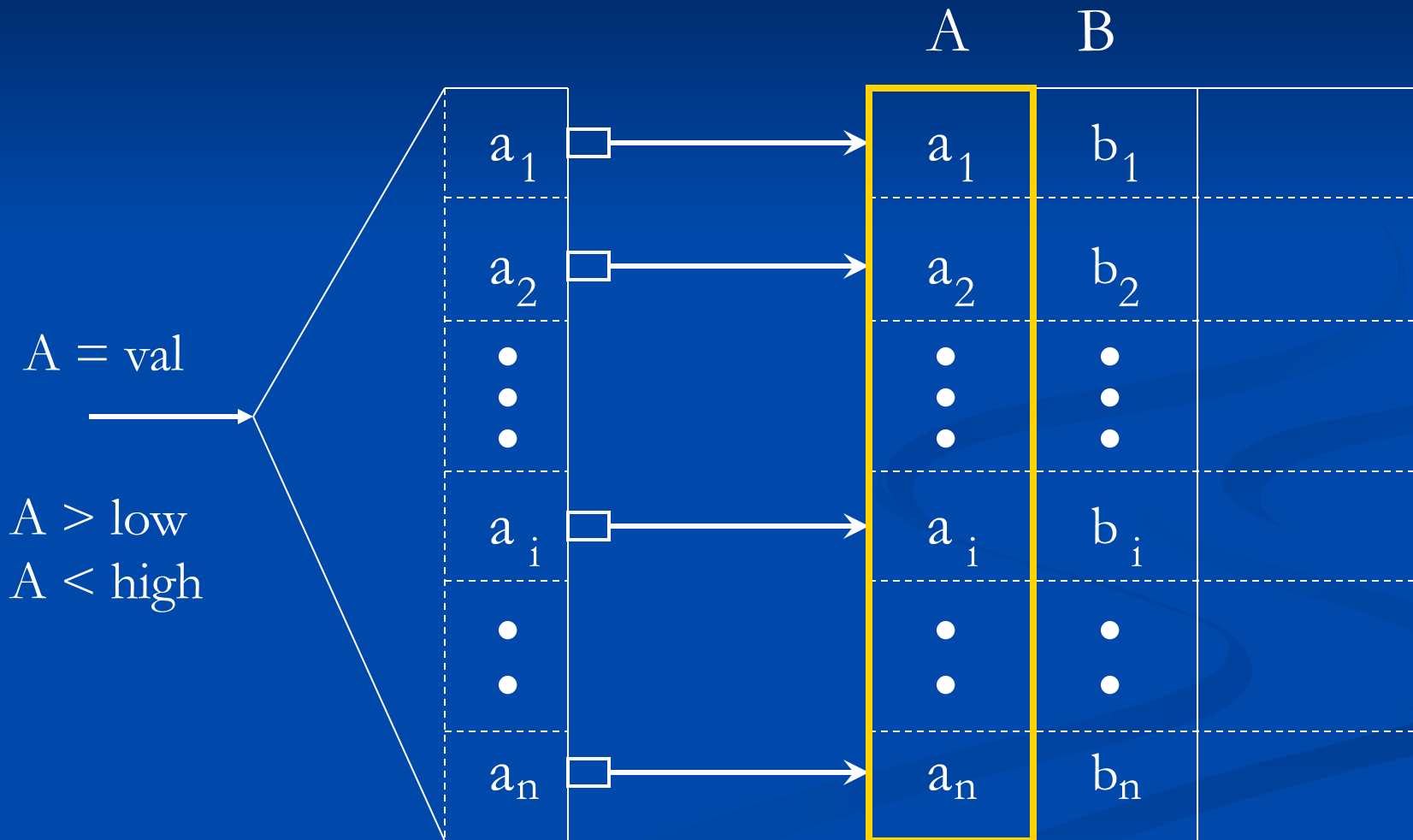
Roadmap

- Motivation
- Single-Attribute Indexes: Overview
- Order-based Indexes
 - B-Trees
- Hash-based Indexes (May cover in future)
 - Extensible Hashing
 - Linear Hashing
- Multi-Attribute Indexes (Chapter 14 GMUW, May cover in future)

Single Attribute Index: General Construction

A	B	
a_1	b_1	
a_2	b_2	
•	•	
•	•	
•	•	
a_i	b_i	
•	•	
•	•	
a_n	b_n	

Single Attribute Index: General Construction

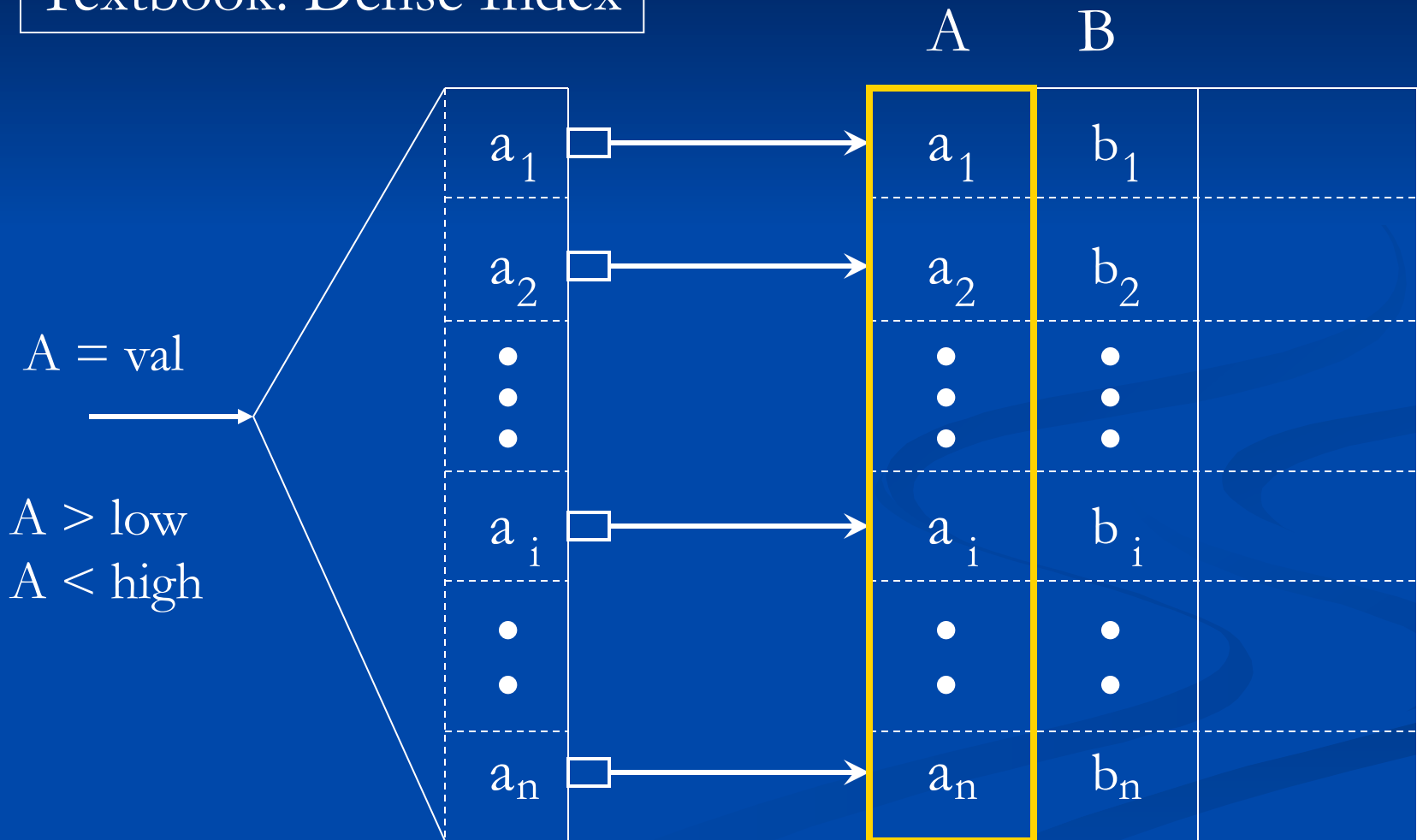


Exceptions

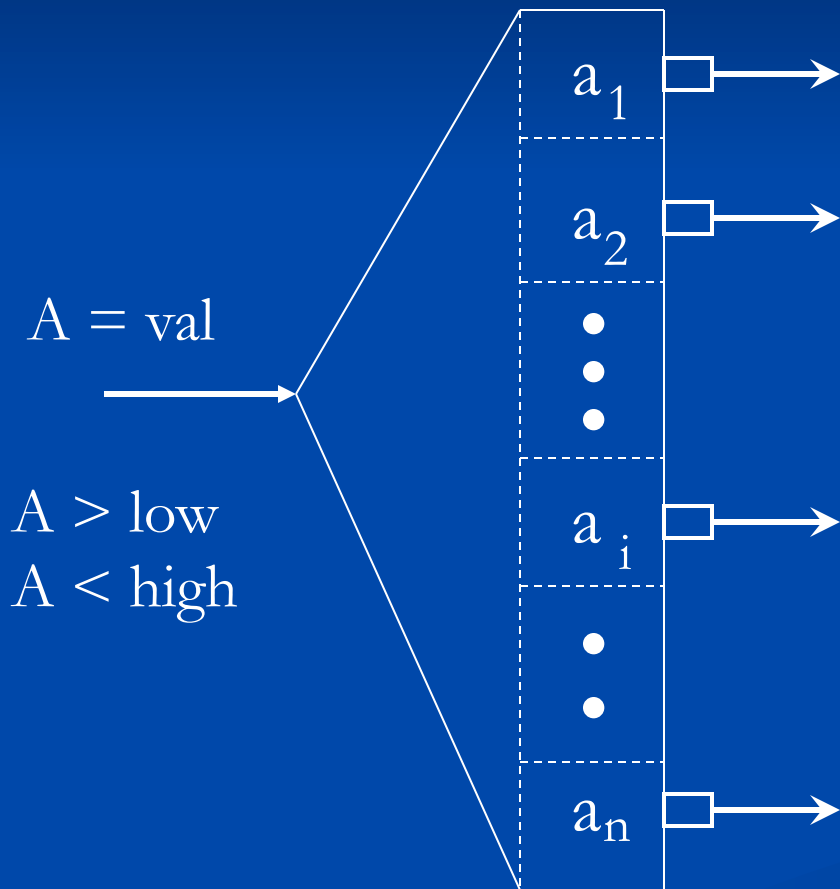
- Sparse Indexes
 - Require specific physical layout of relation
 - Example: Relation sorted on indexed attribute
 - More efficient

Single Attribute Index: General Construction

Textbook: Dense Index



Single Attribute Index: General Construction



How do we organize
(attribute, pointer) pairs?

Idea: Use dictionary
data structures

Issue: Disk resident?

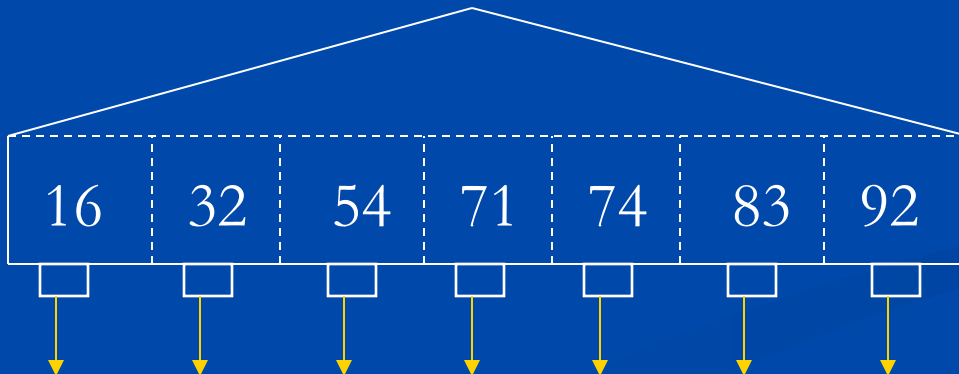
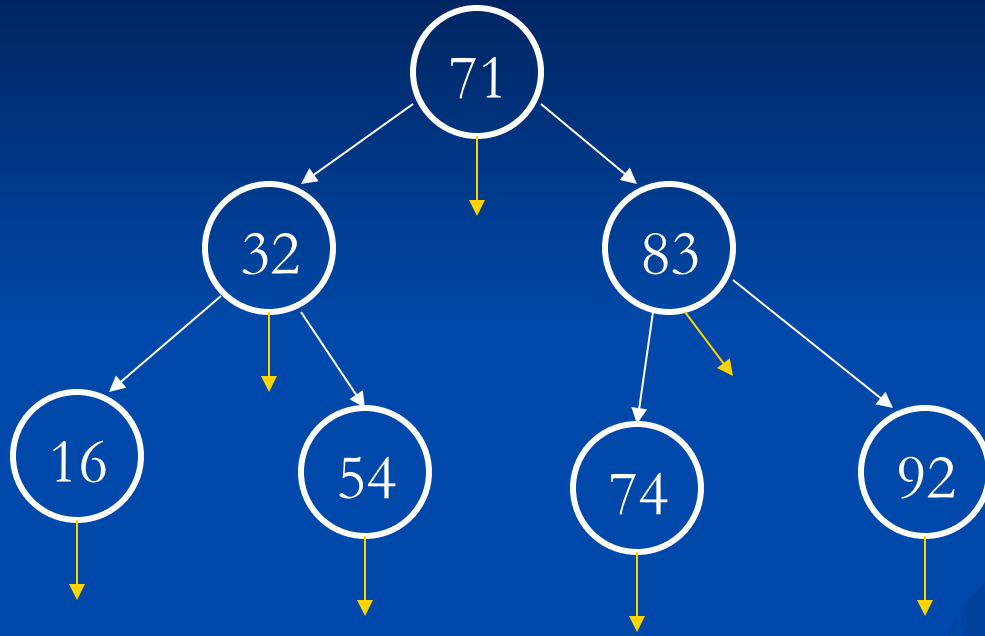
Roadmap

- Motivation
- Single-Attribute Indexes: Overview
- Order-based Indexes
 - B-Trees
- Hash-based Indexes
 - Extensible Hashing
 - Linear Hashing
- Multi-Attribute Indexes

B-Trees

- Adaptation of search tree data structure
 - 2-3 trees
- Supports range predicates (and equality)

Use Binary Search Tree Directly?



Use Binary Search Tree Directly?

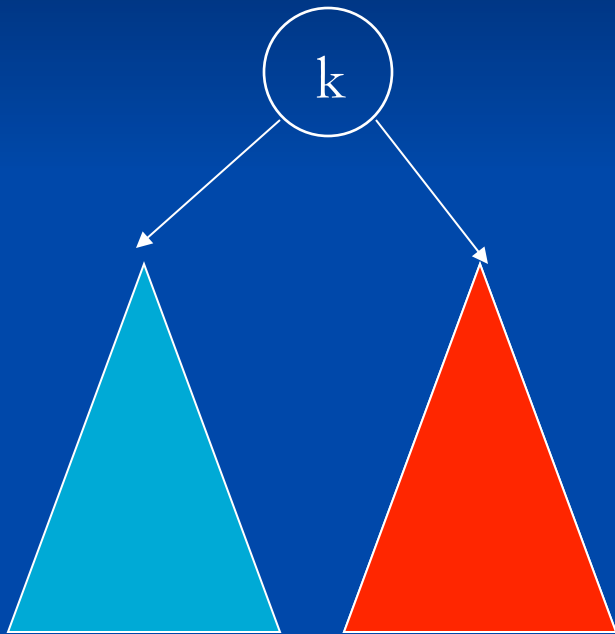
- Store records of type
<key, left-ptr, right-ptr, data-ptr>
- Remember position of root
- Question: will this work?
 - Yes
 - But we can do better!

Use Binary Search Tree Directly?

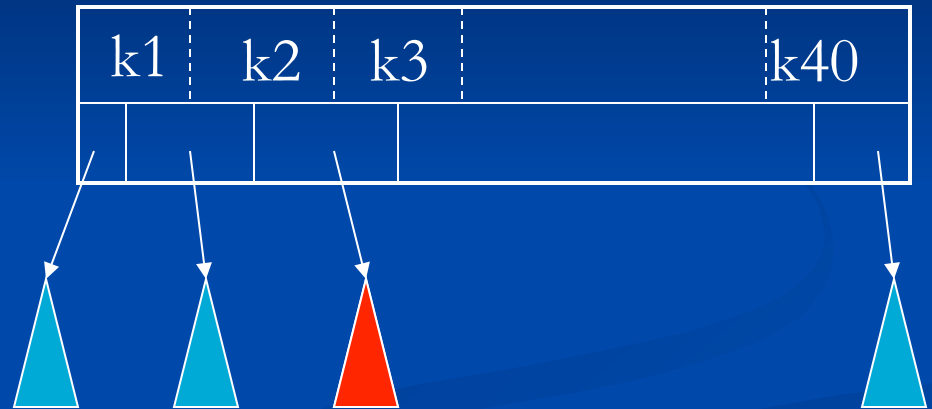
- Number of keys: 1 M
- Number of levels: $\log(2^{20}) = 20$
- Total cost index lookup: 20 random disk I/O
 - $20 \times 20 \text{ ms} = 400 \text{ ms}$

B-Tree: less than 3 random disk I/O

B-Tree vs. Binary Search Tree

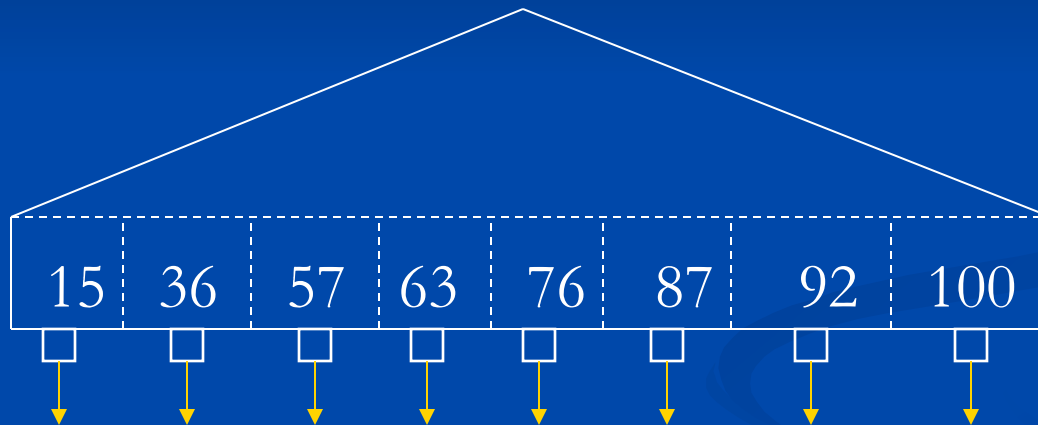


1 Random I/O prunes tree by half

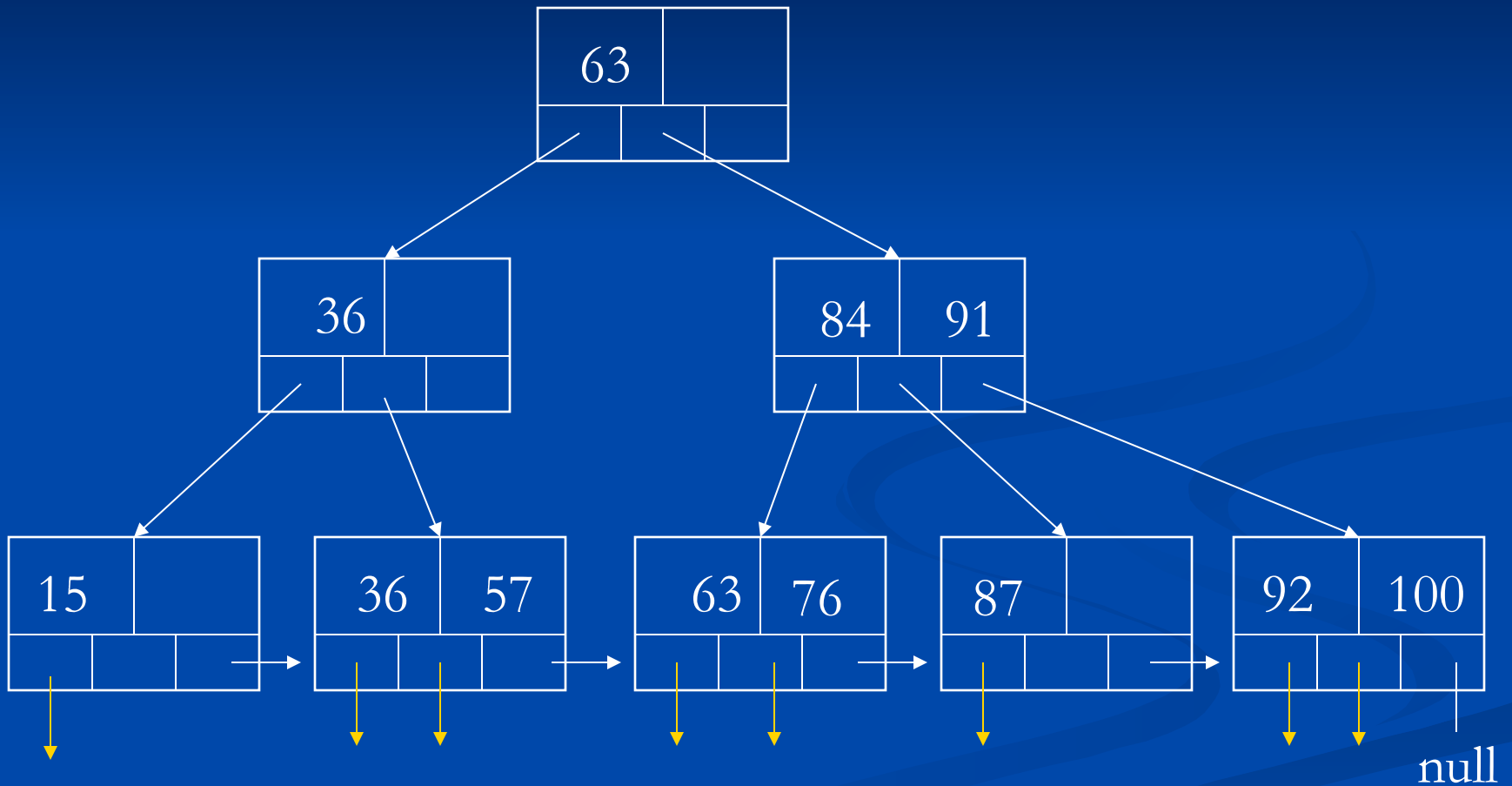


1 Random I/O prunes tree by 40

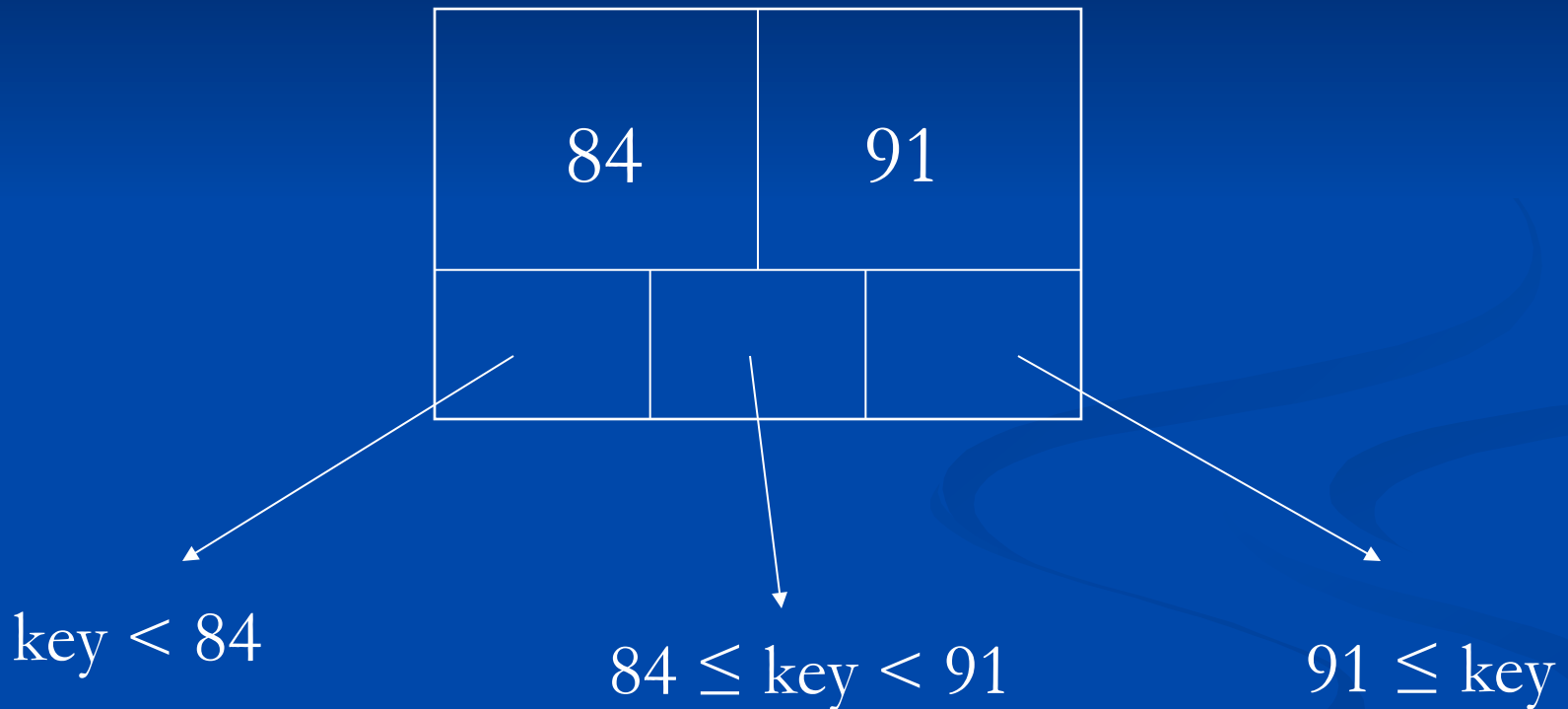
B-Tree Example



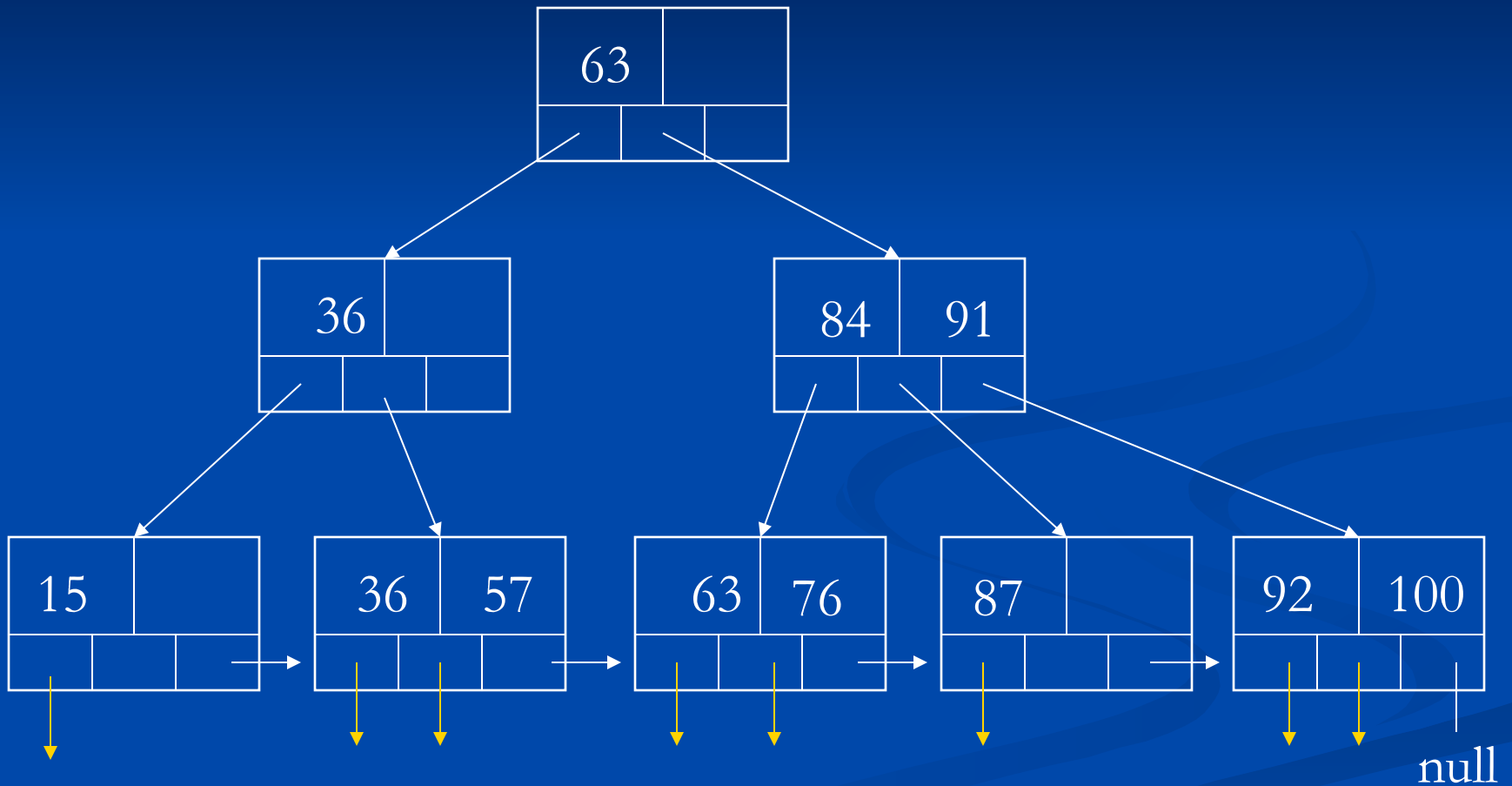
B-Tree Example



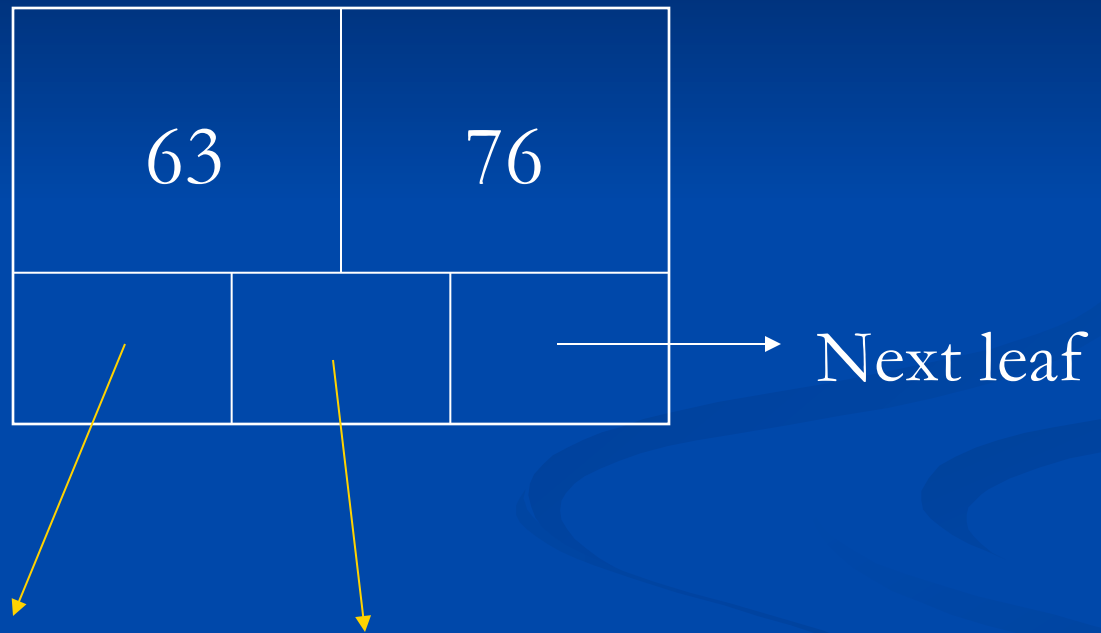
Meaning of Internal Node



B-Tree Example



Meaning of Leaf Nodes

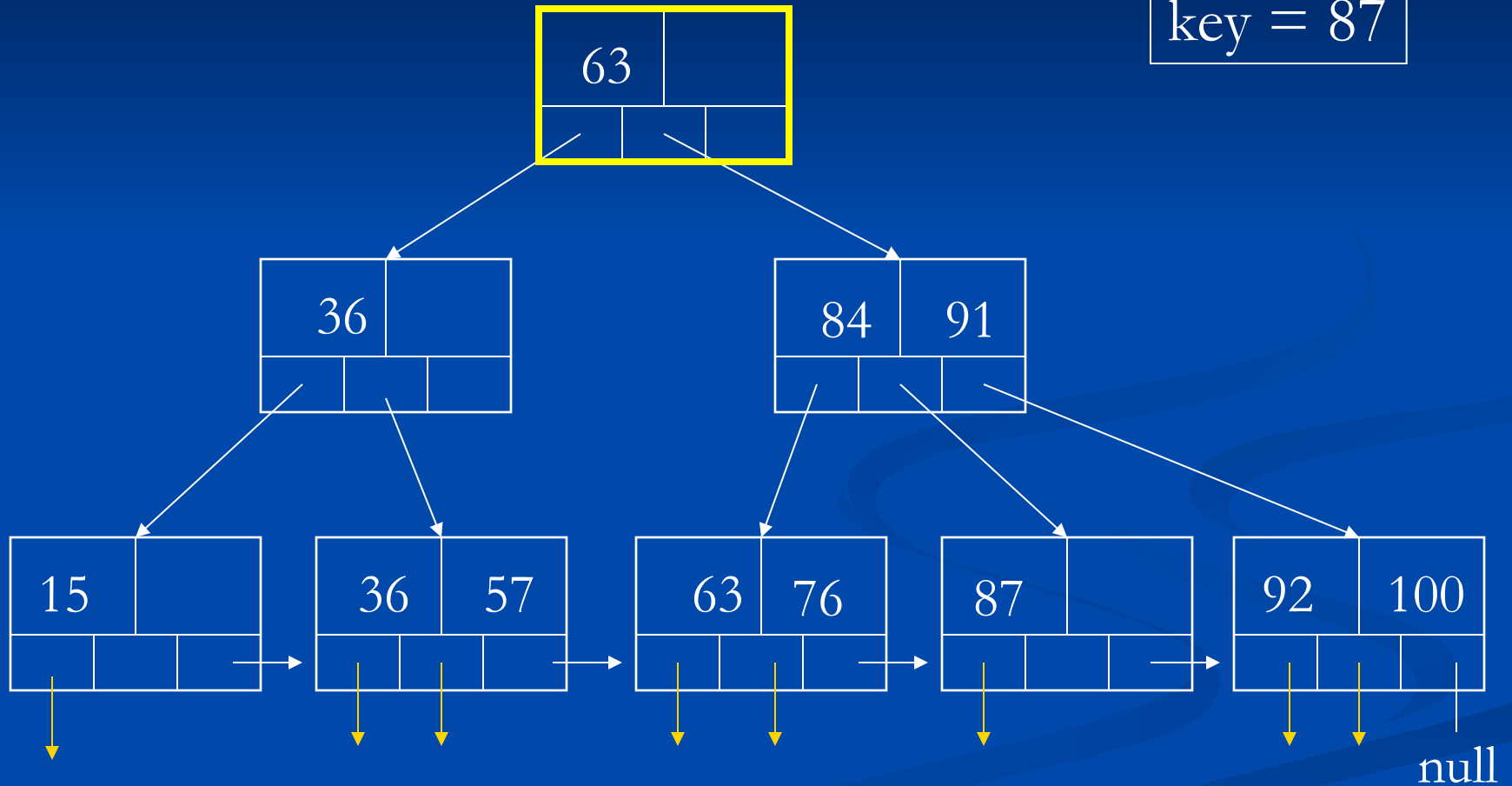


pointer to record 63

pointer to record 76

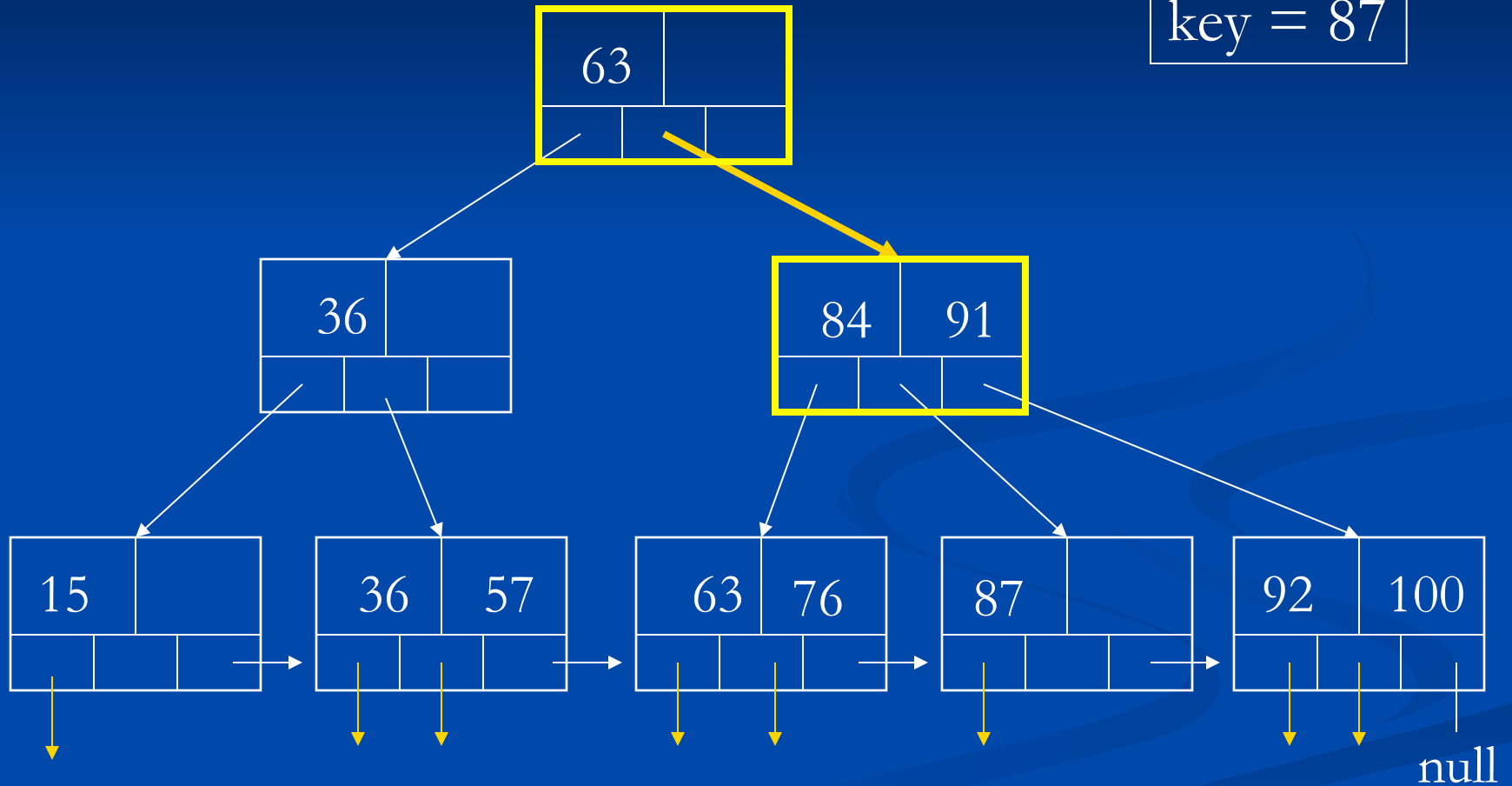
Equality Predicates

key = 87



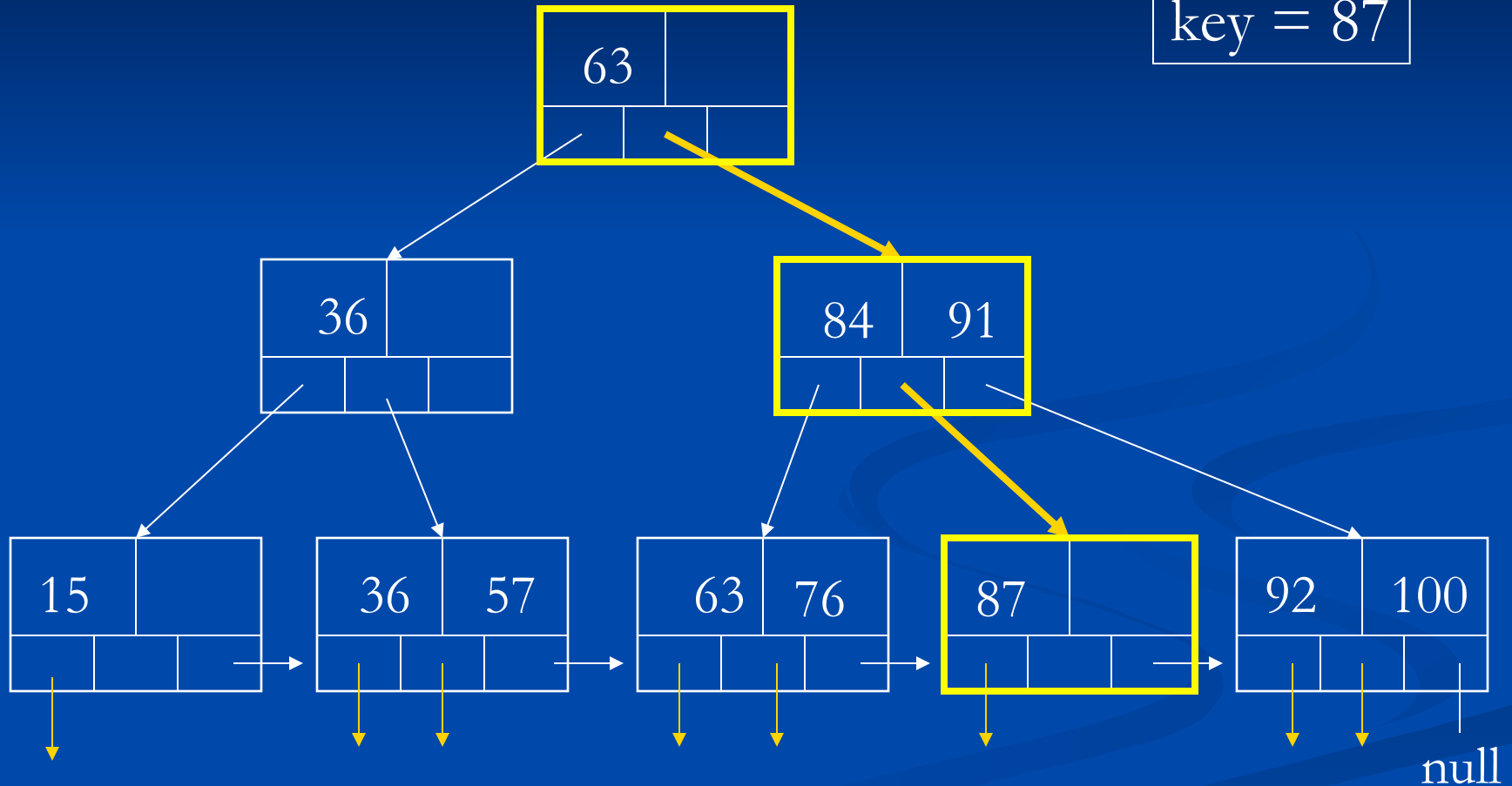
Equality Predicates

key = 87



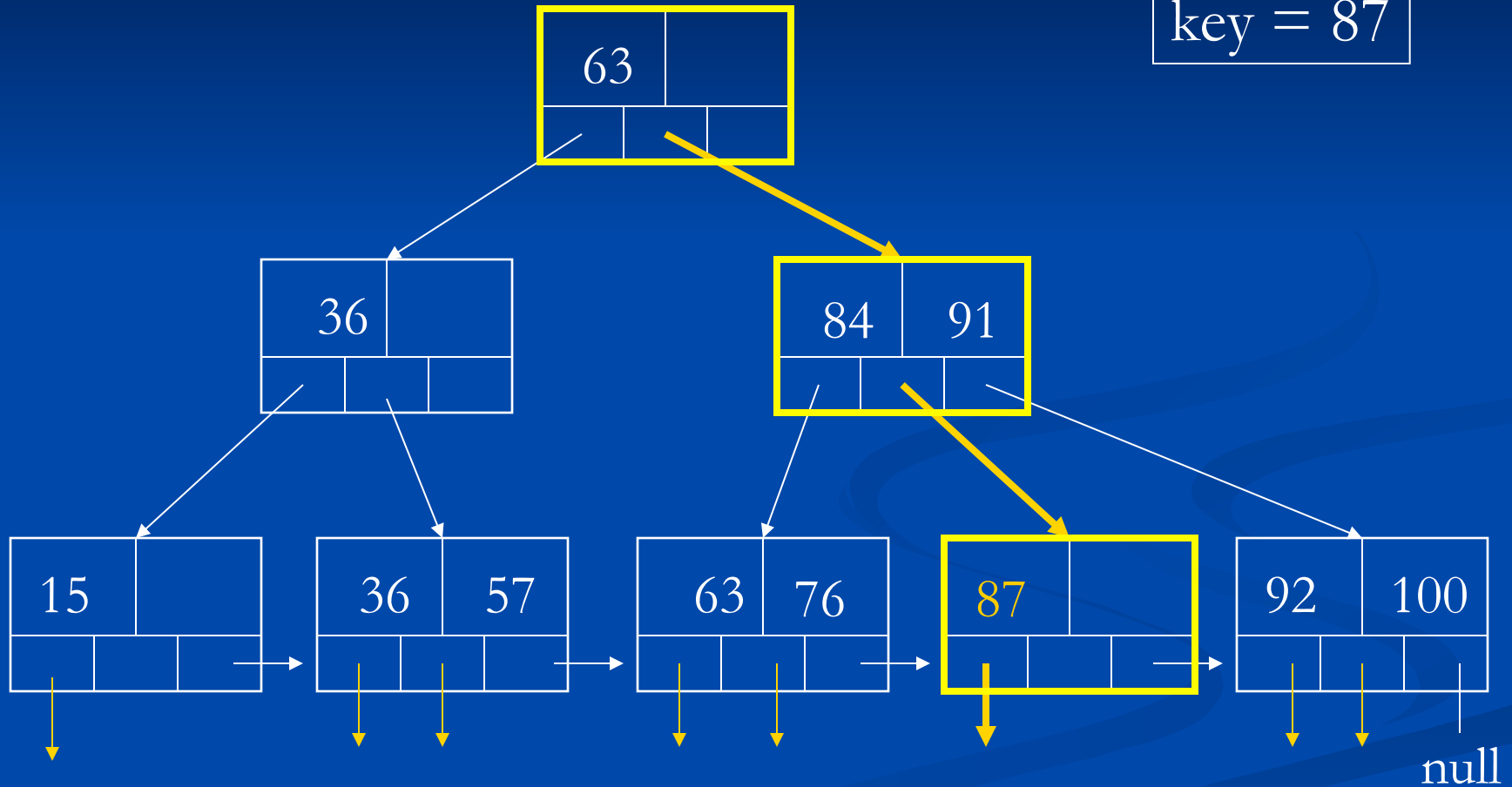
Equality Predicates

key = 87



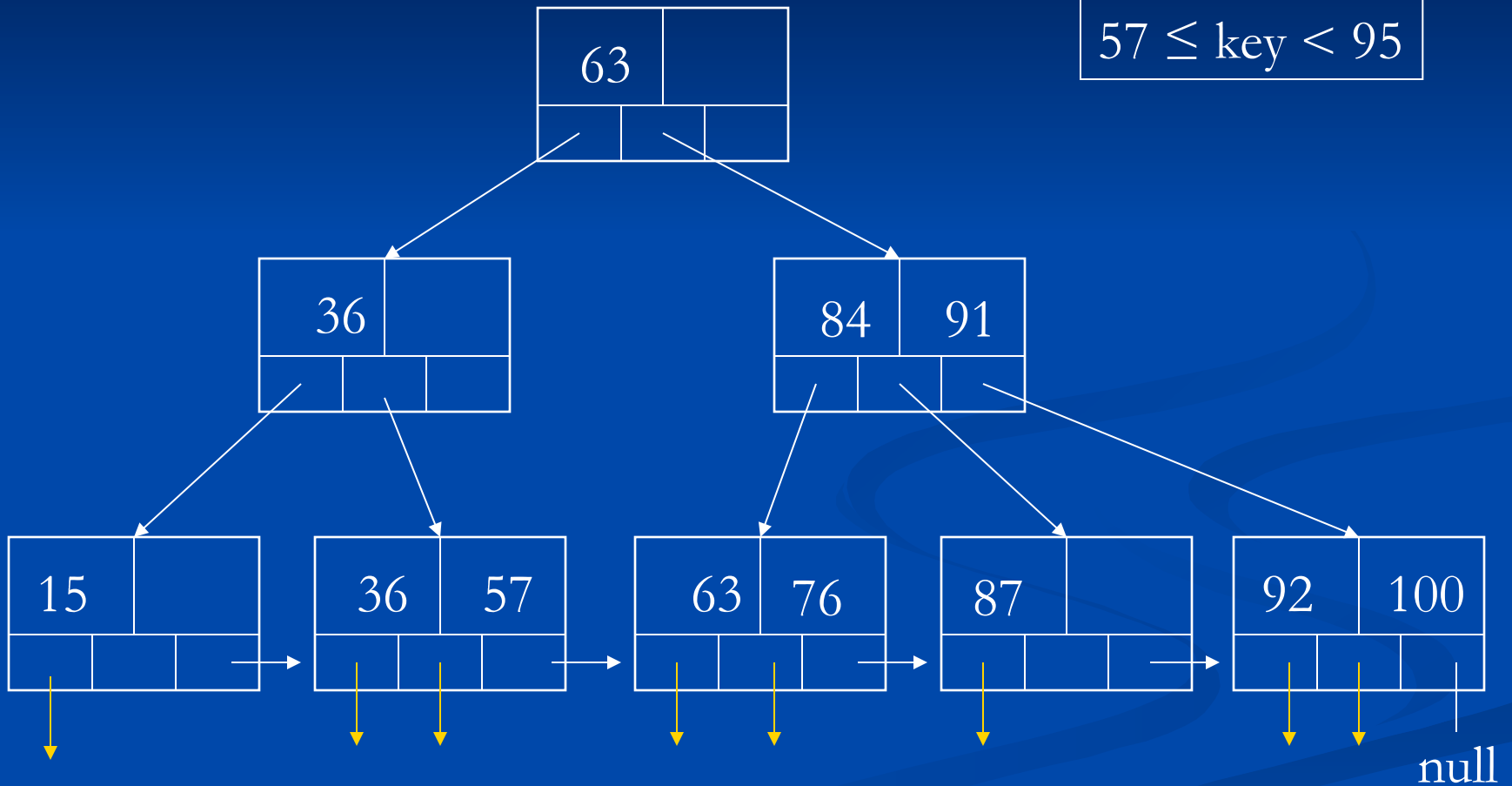
Equality Predicates

key = 87

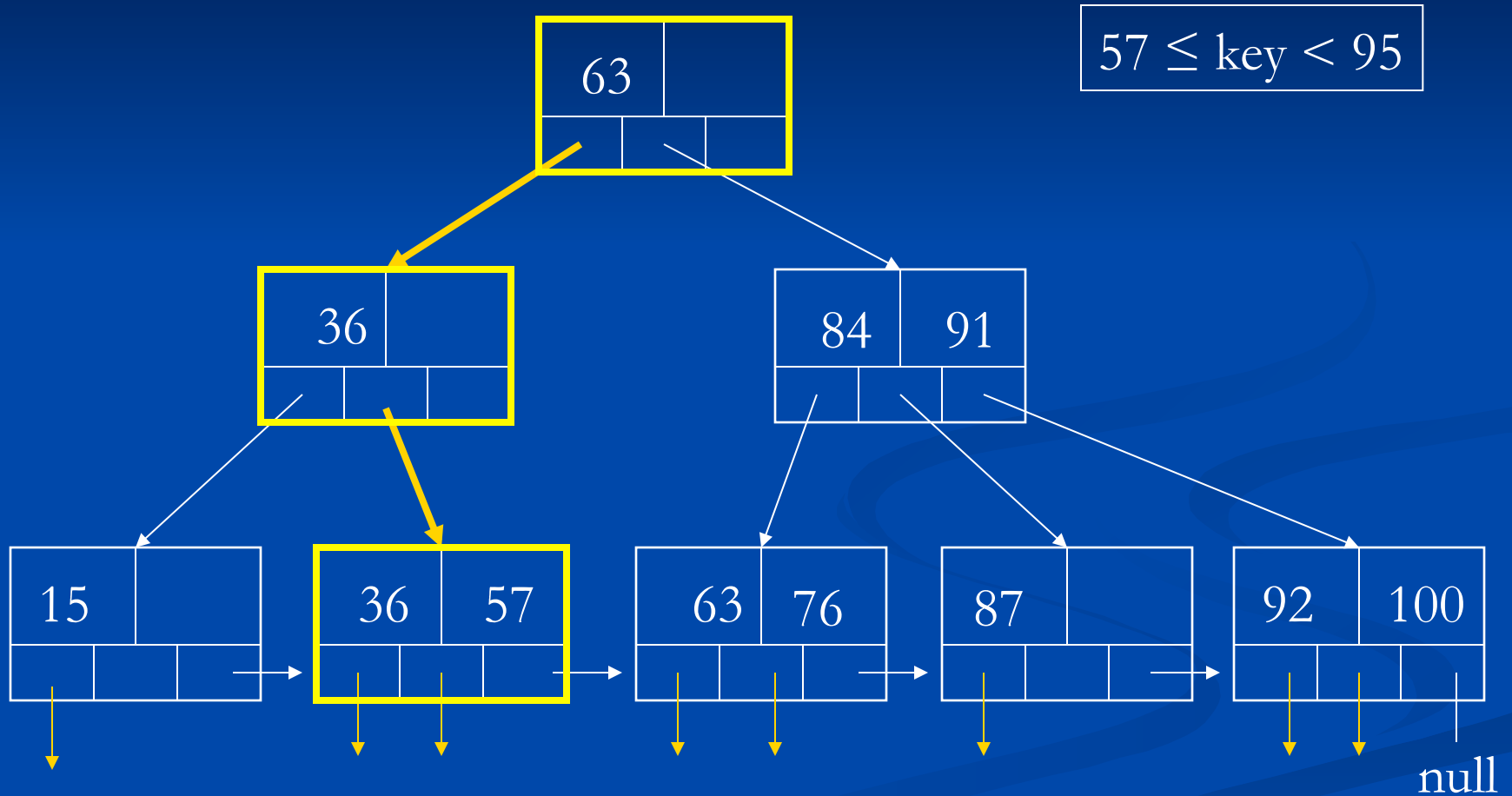


Range Predicates

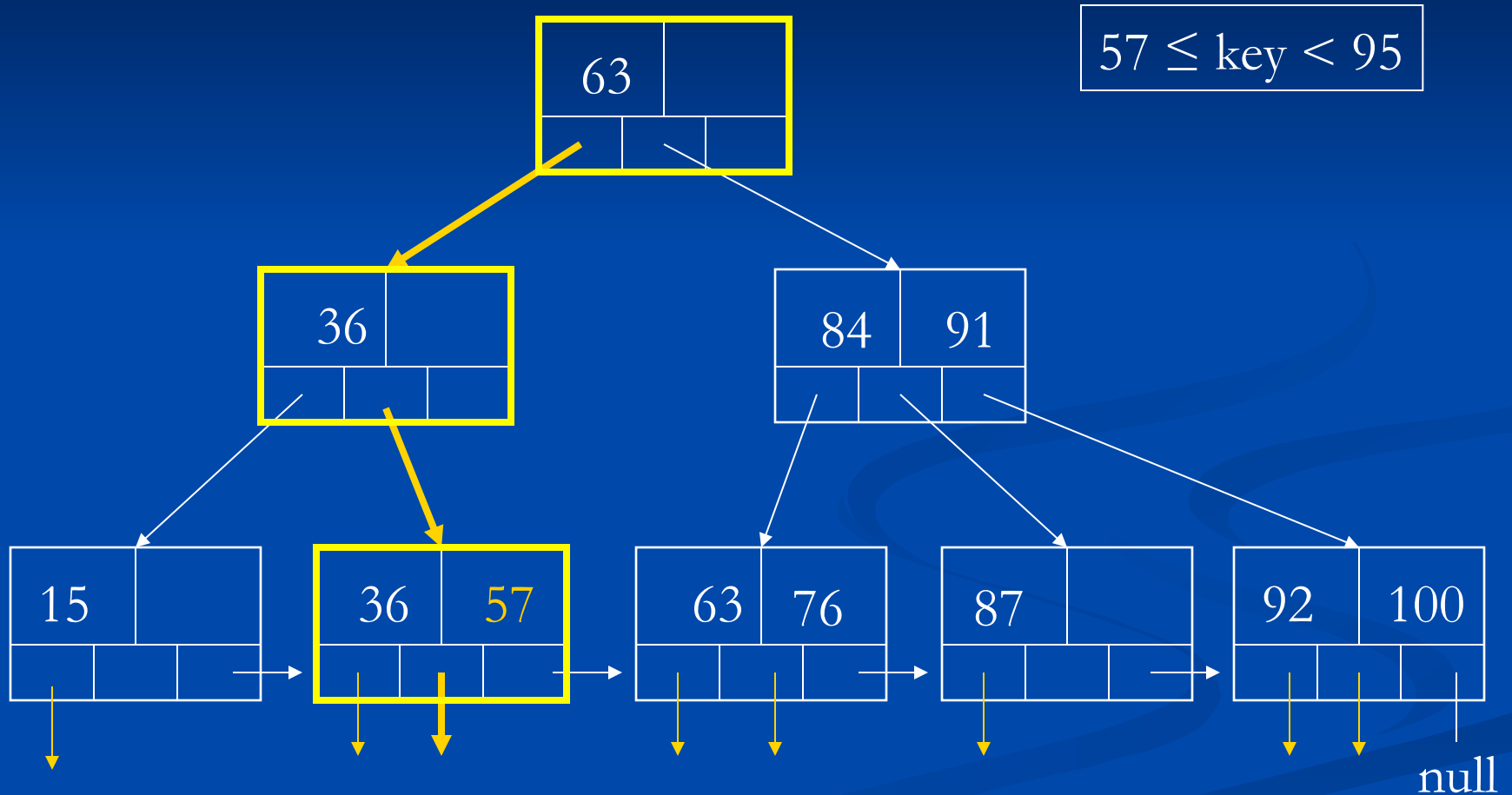
$57 \leq \text{key} < 95$



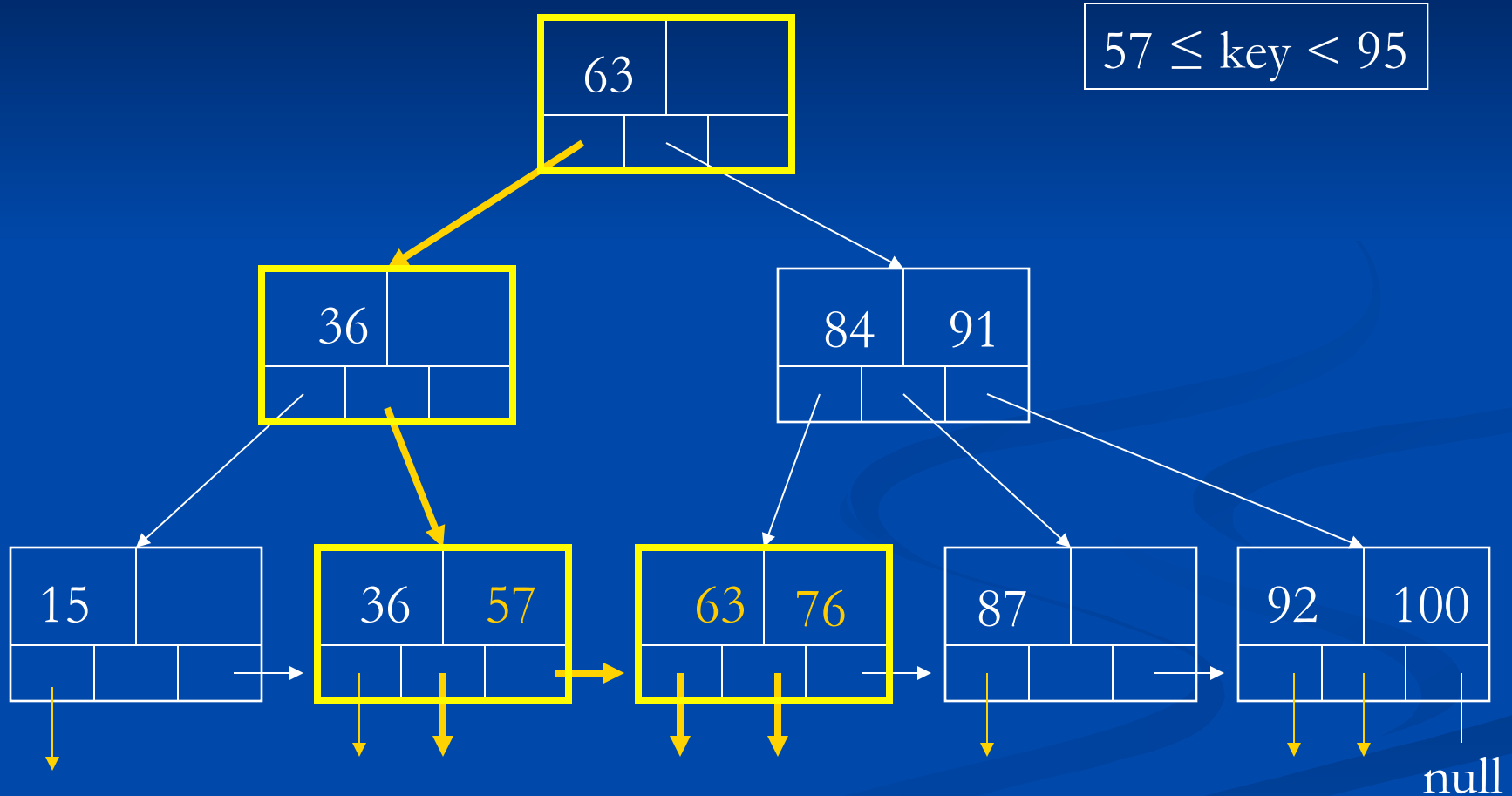
Range Predicates



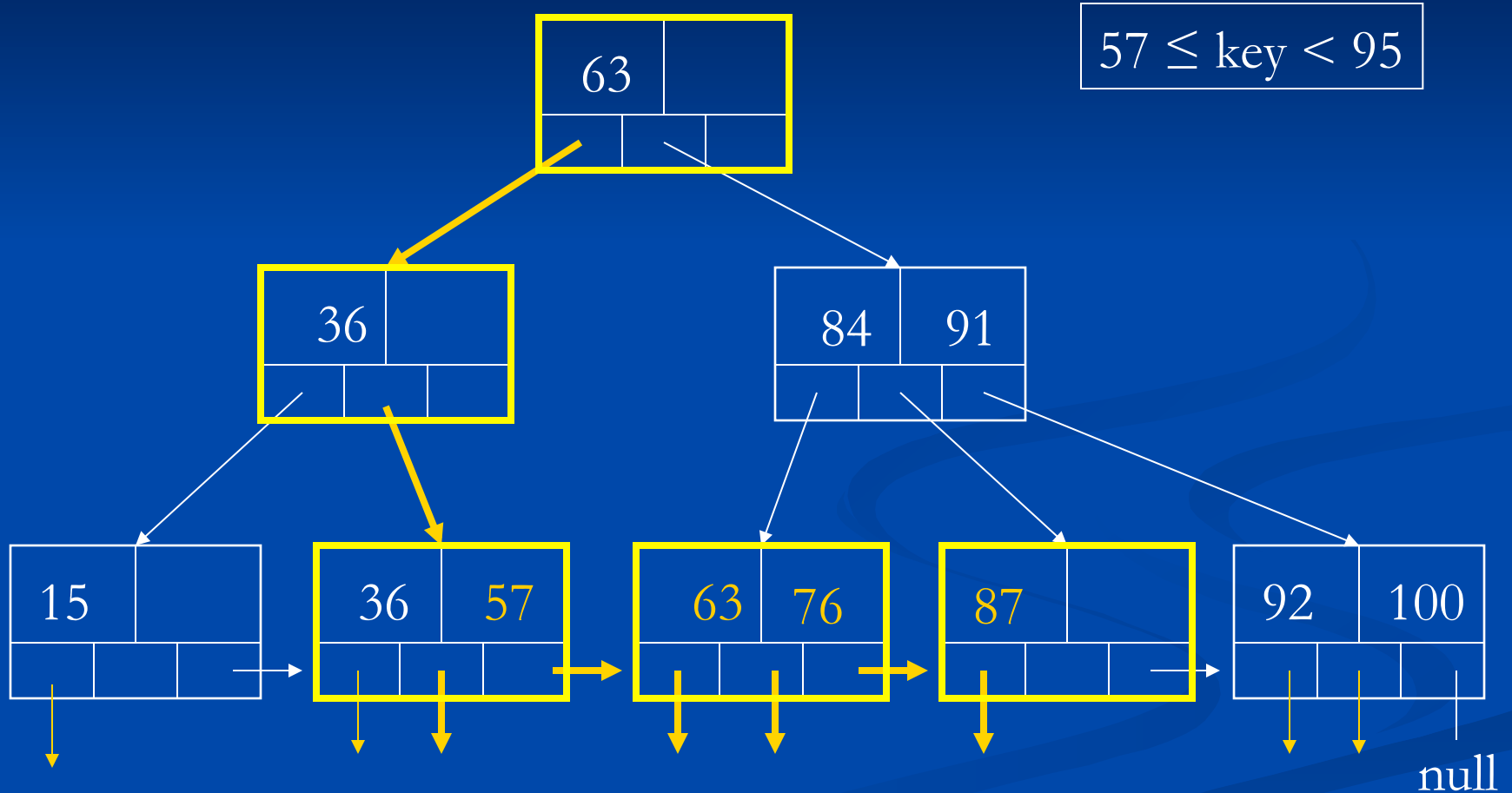
Range Predicates



Range Predicates

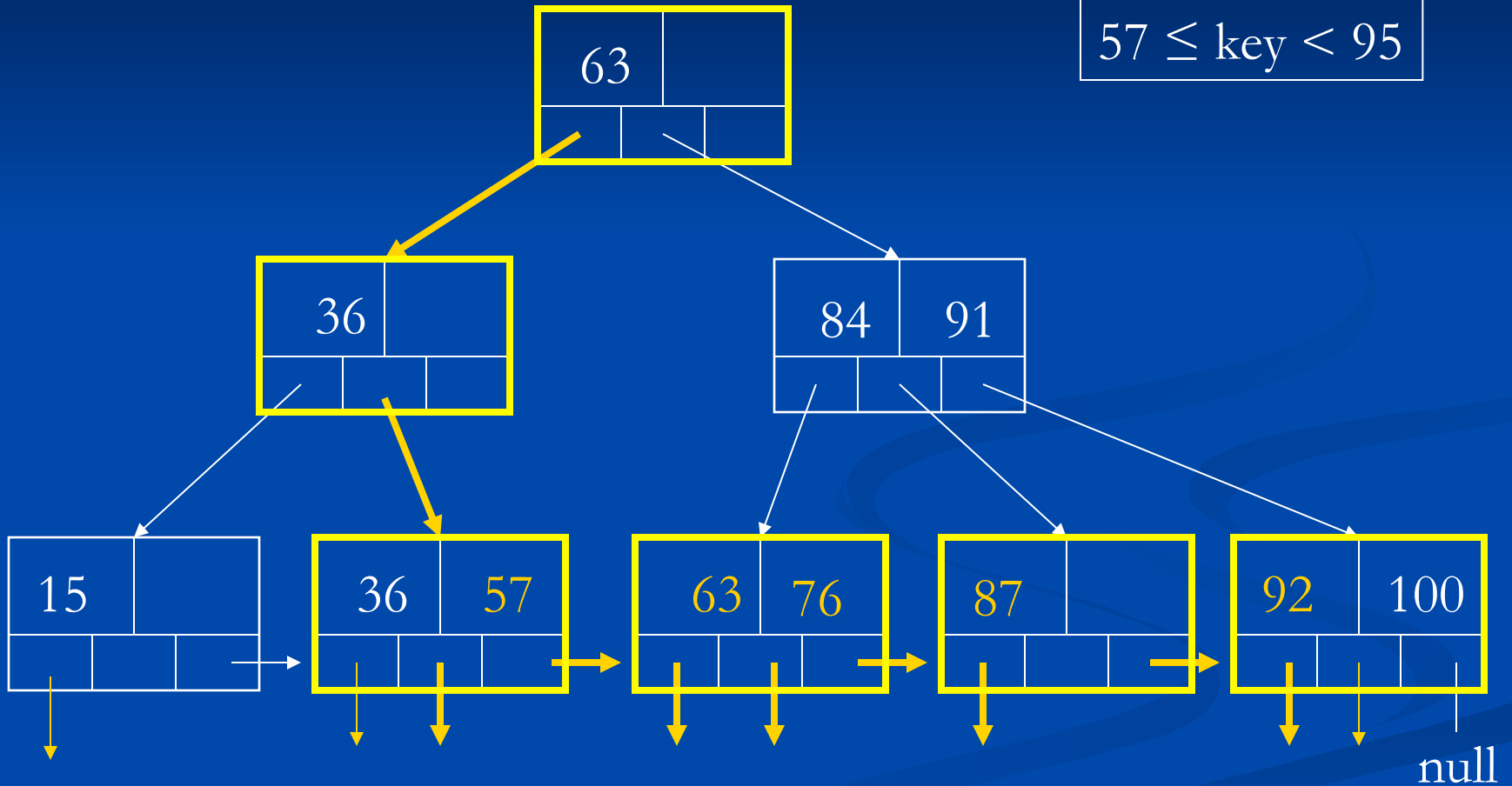


Range Predicates



Range Predicates

$57 \leq \text{key} < 95$

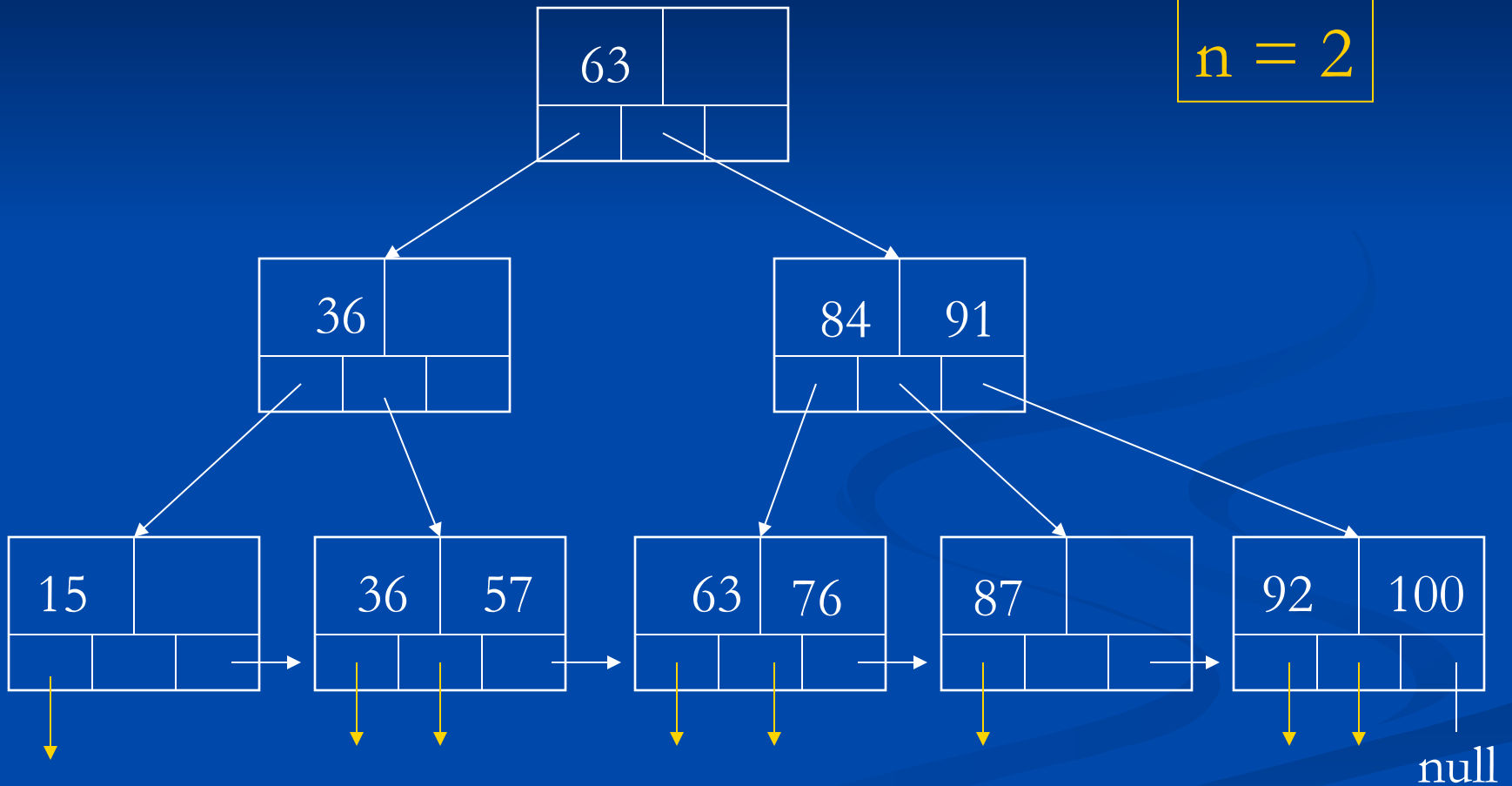


General B-Trees

- Fixed parameter: n
- Number of keys: n
- Number of pointers: $n + 1$

B-Tree Example

$n = 2$

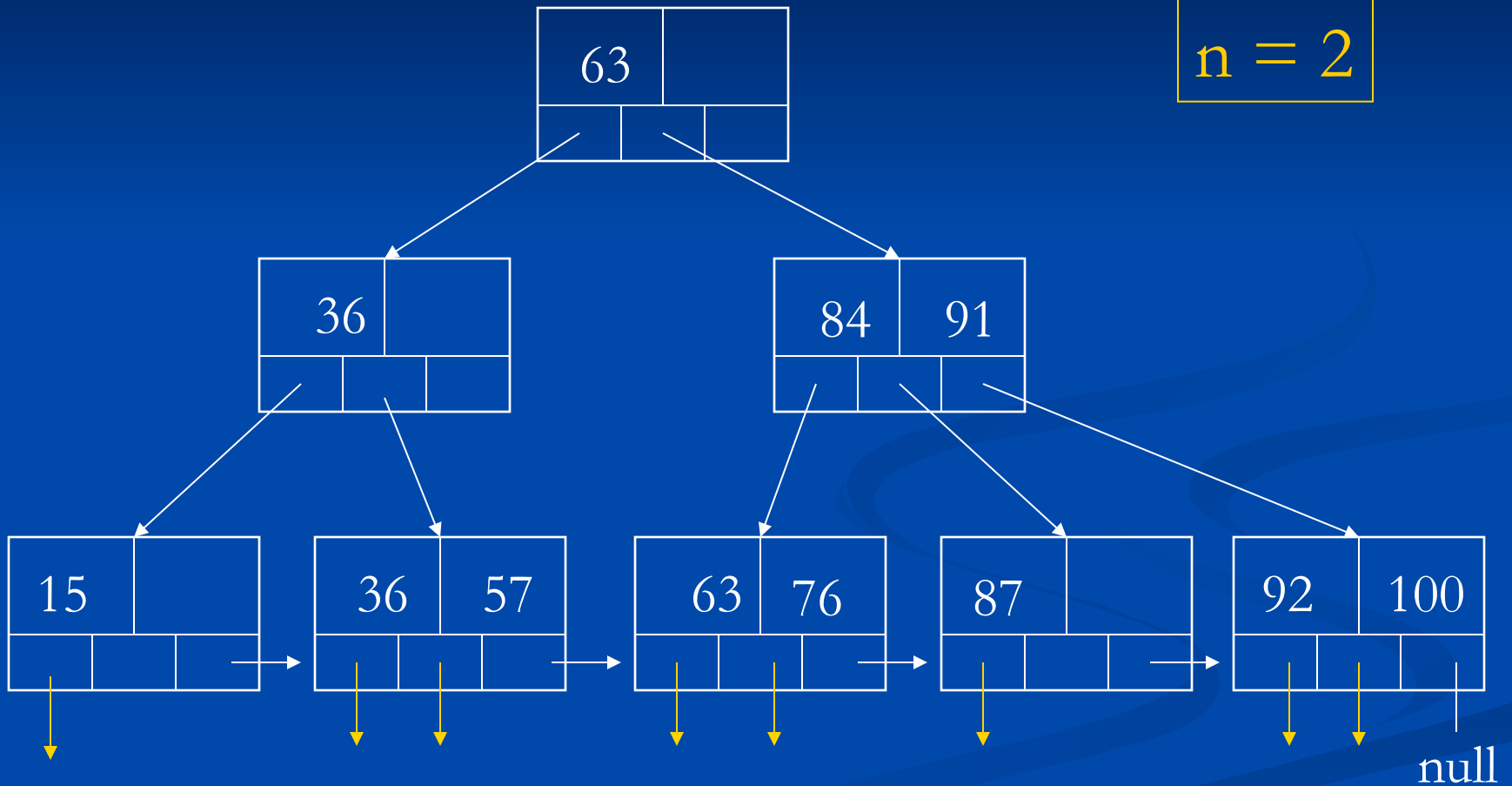


General B-Trees

- Fixed parameter: n
- Number of keys: n
- Number of pointers: $n + 1$
- All leaves at same depth
- All (key, record pointer) in leaves

B-Tree Example

$n = 2$



General B-Trees:

Space related constraints

- Use at least

Root: 2 pointers

Internal: $\lceil (n+1)/2 \rceil$ pointers

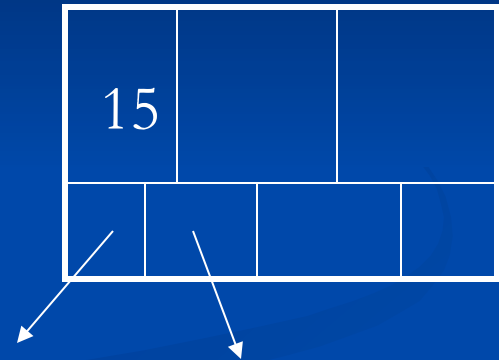
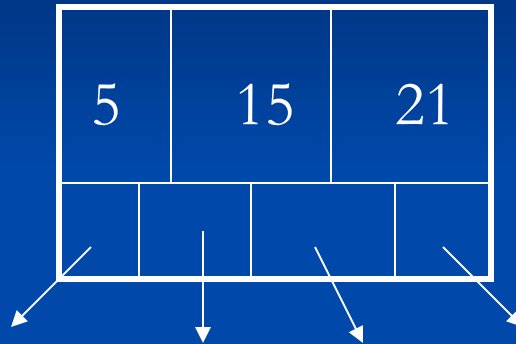
Leaf: $\lfloor (n+1)/2 \rfloor$ pointers to data

n=3

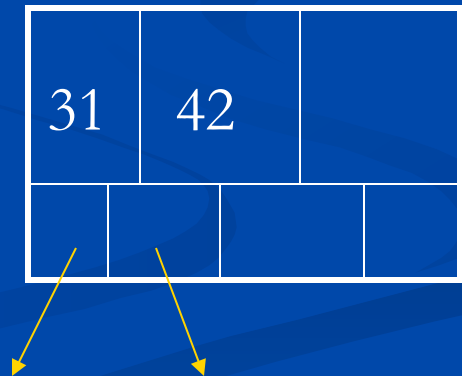
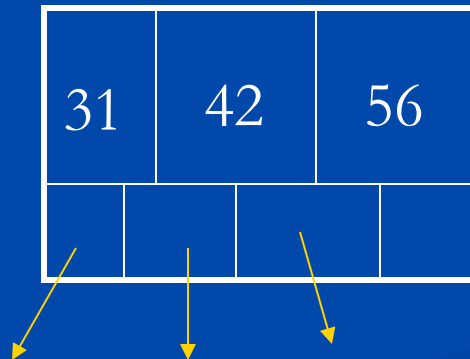
Max

Min

Internal



Leaf



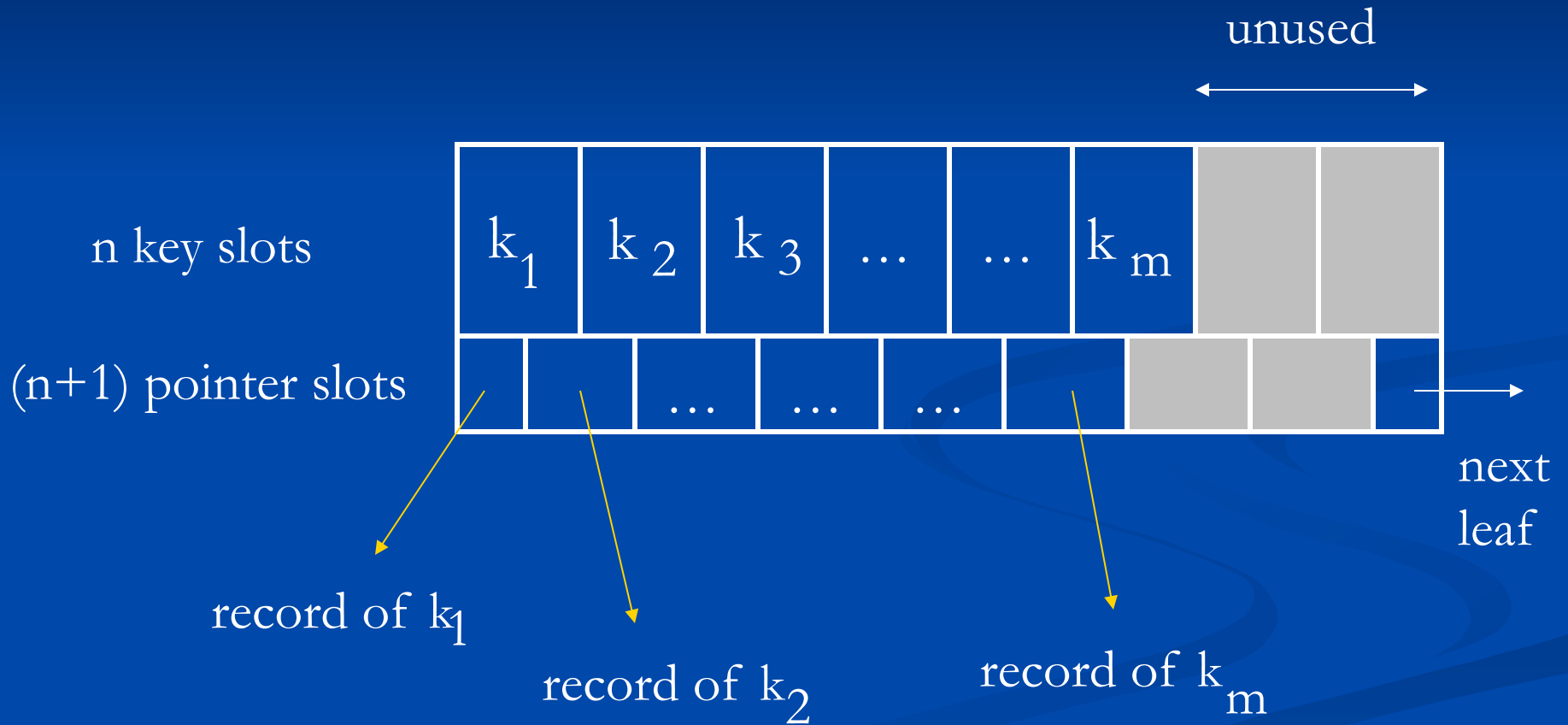
Leaf Nodes

n key slots



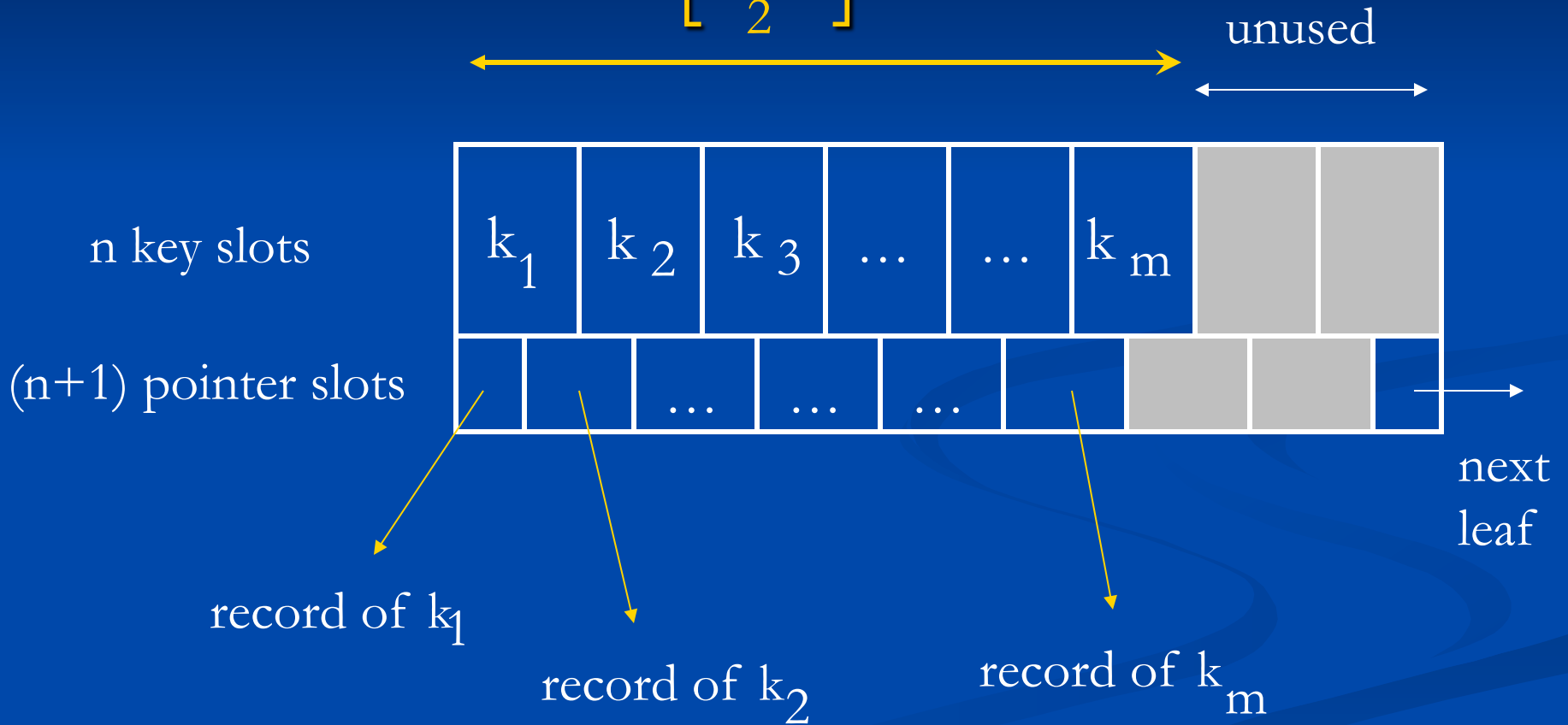
(n+1) pointer slots

Leaf Nodes



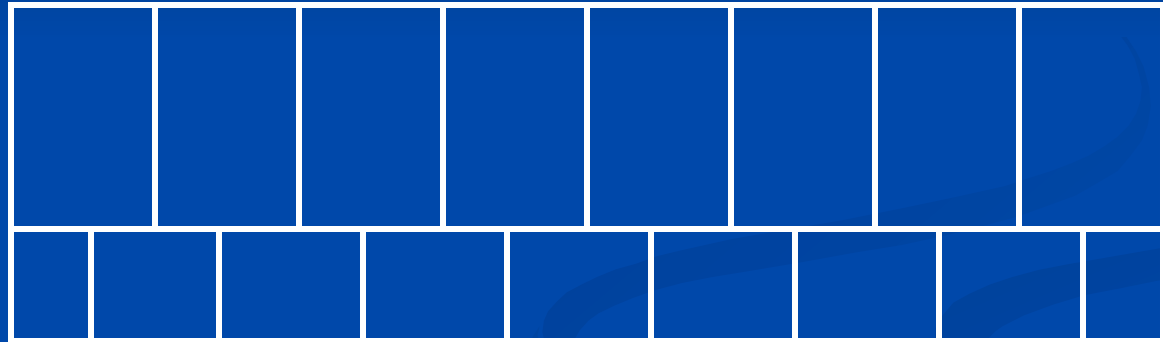
Leaf Nodes

$$m \geq \left\lceil \frac{(n+1)}{2} \right\rceil$$



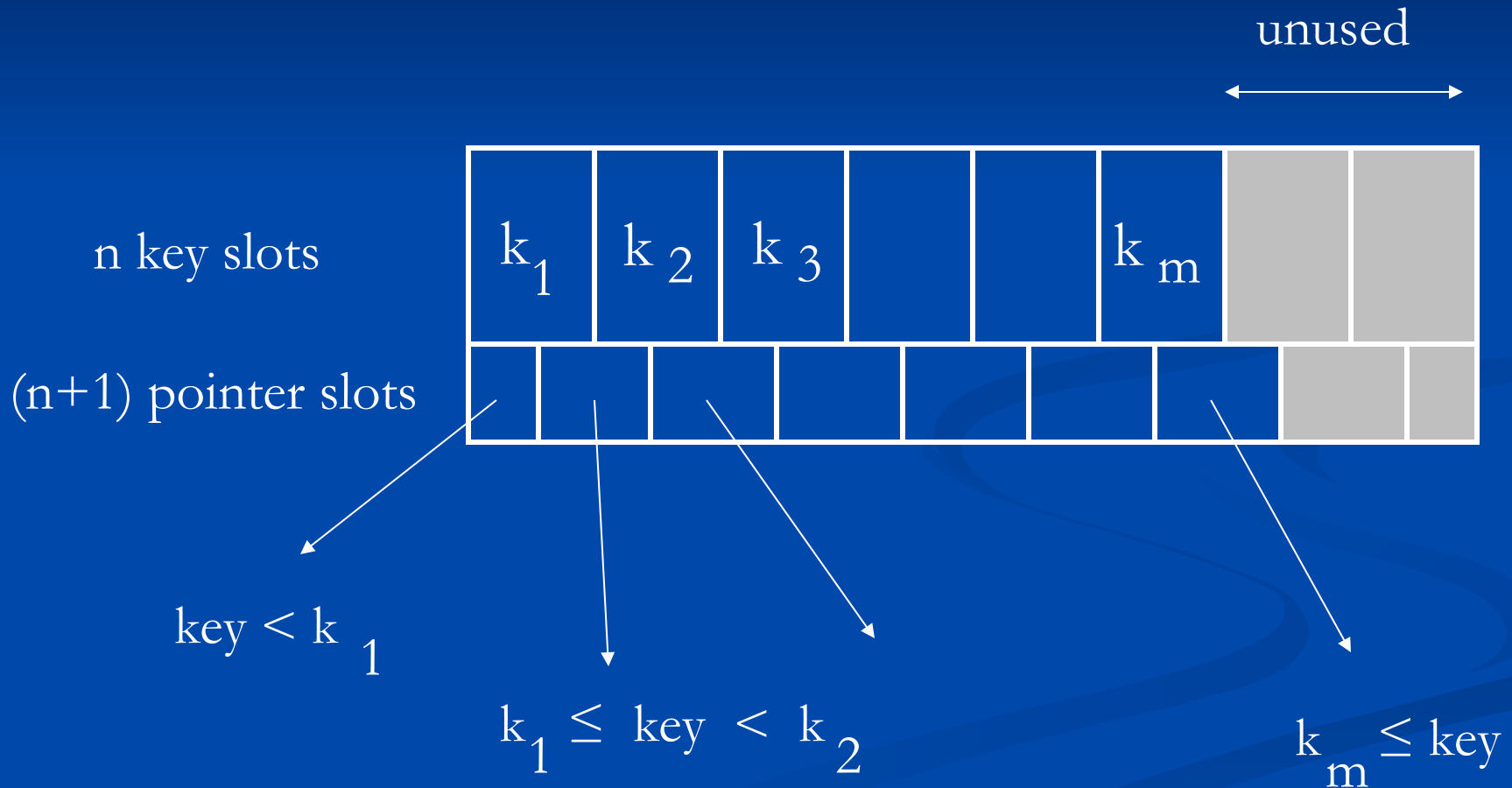
Internal Nodes

n key slots

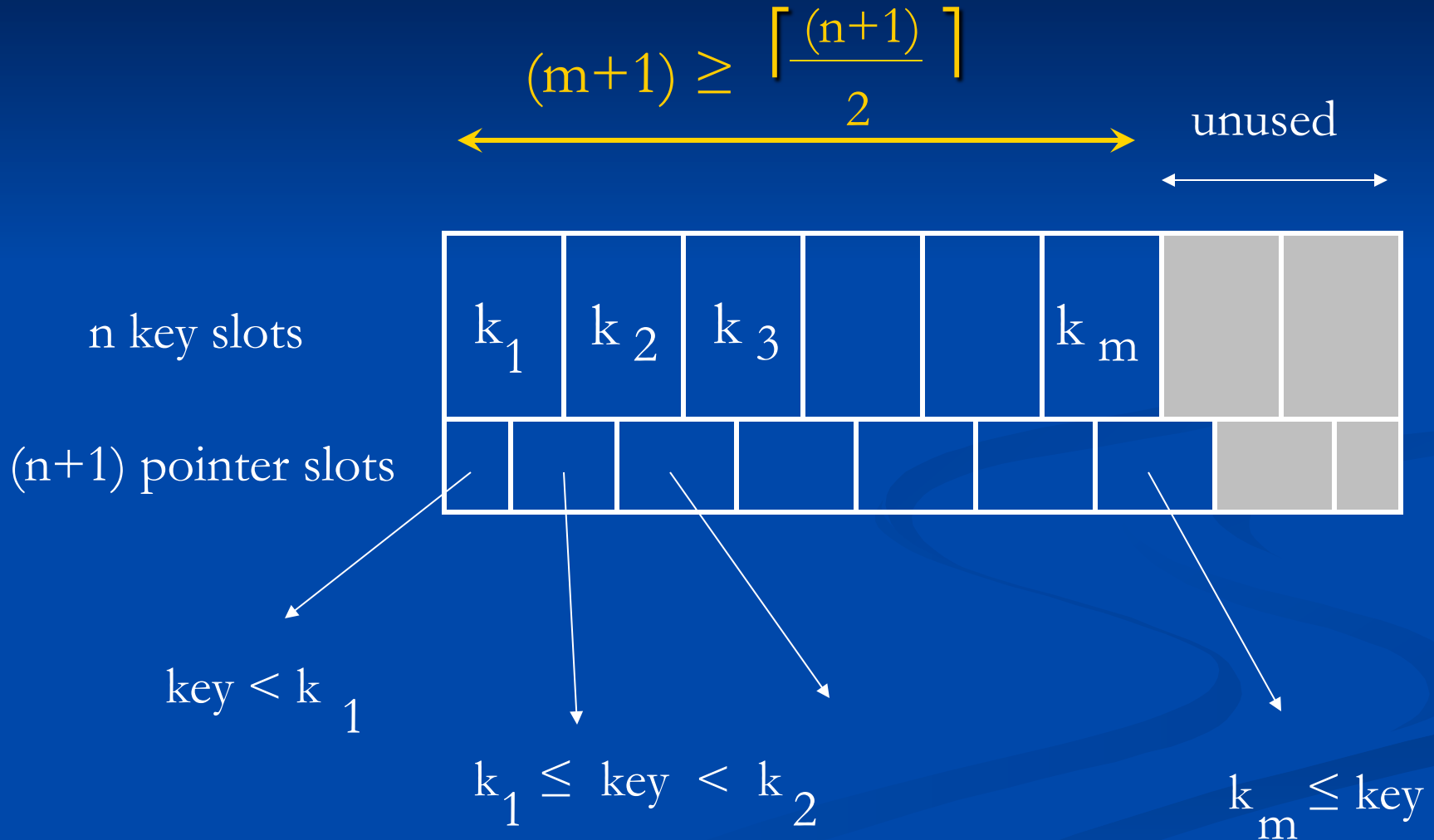


(n+1) pointer slots

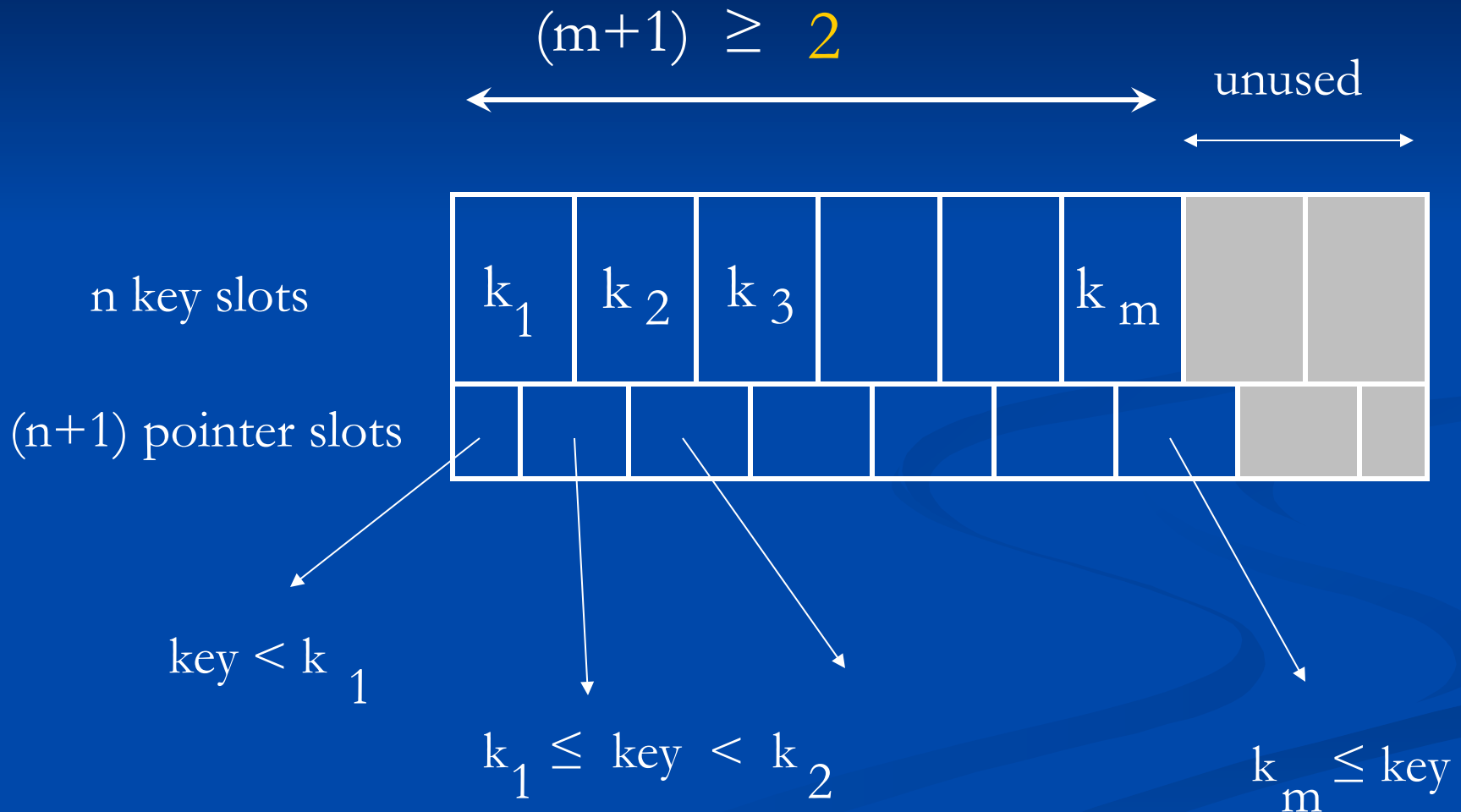
Internal Nodes



Internal Nodes



Root Node



Limits

- Why the specific limits
 $\lceil (n+1)/2 \rceil$ and $\lfloor (n+1)/2 \rfloor$?
- Why different limits for leaf and internal nodes?
- Can we reduce each limit?
- Can we increase each limit?
- What are the implications?