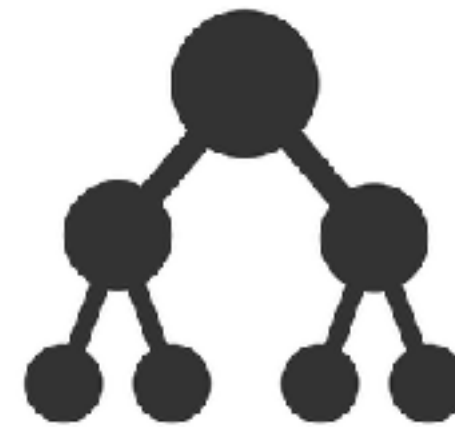


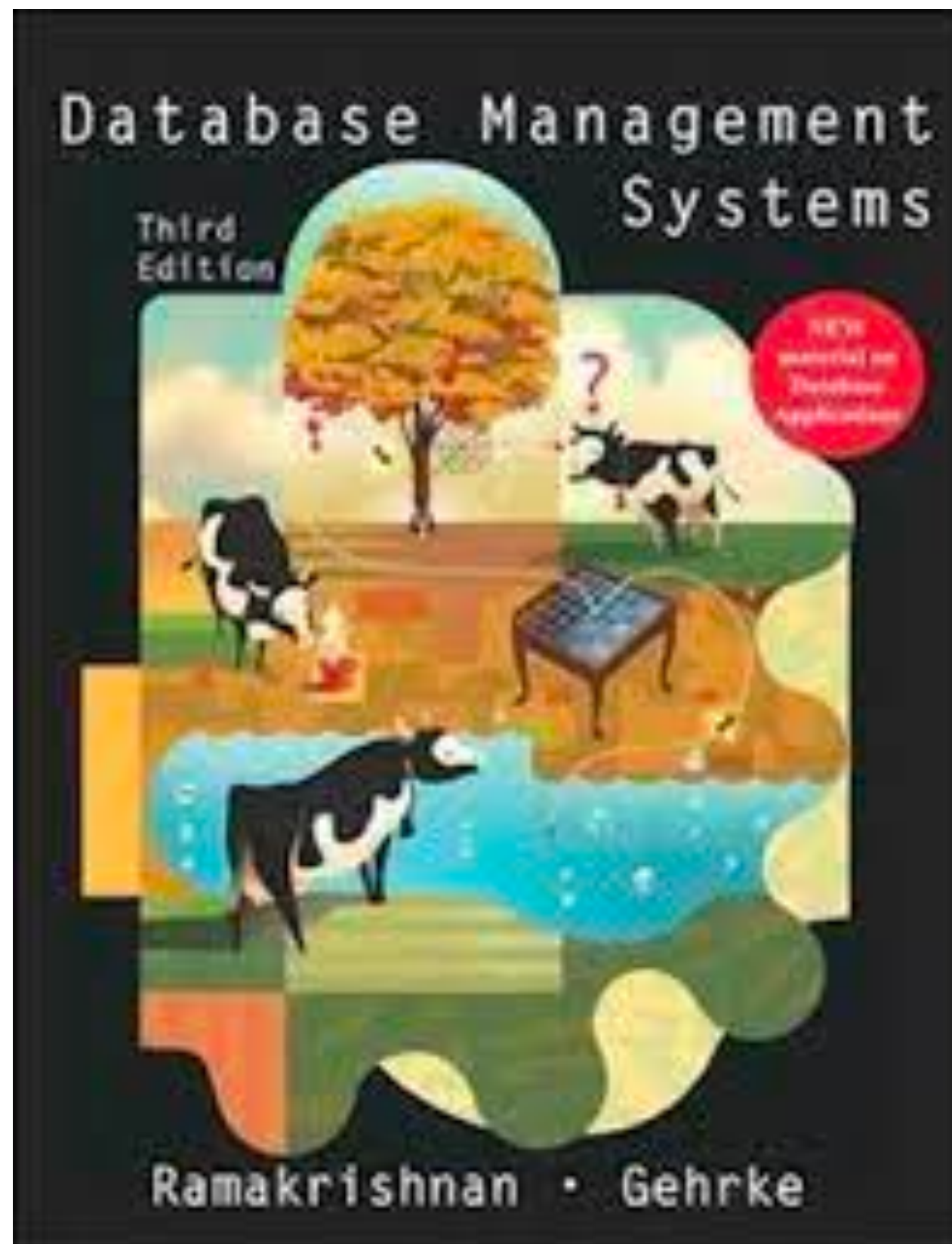
Indexing



Database System Technology

Niv Dayan

We'll start at 3:10 pm



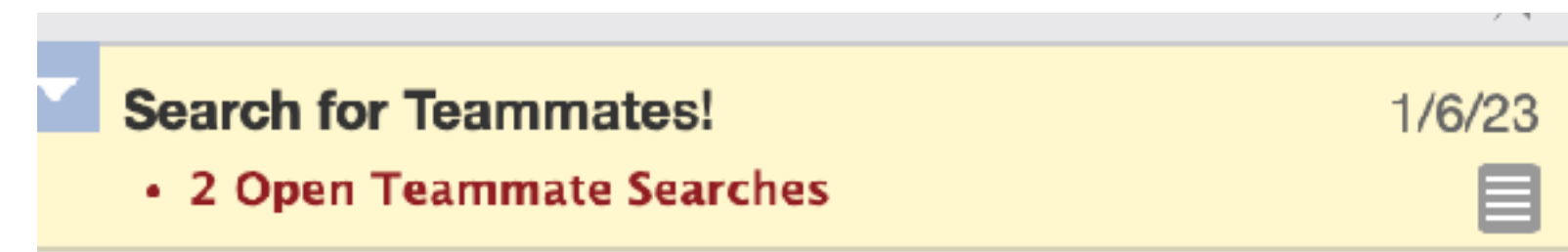
Chapter 8 - Overview of Storage & Indexing

Chapter 10 - Tree-Structured Indexing

Chapter 11 - Hash-Based Indexing

Groups

We'll ask you to register them soon and group up anyone who is still group-less.



A B C

Metadata		
Data	Data	Data
...	...	...
Data	Data	Data
Metadata		
Data	Data	Data
...	...	...
Data	Data	Data
Metadata		
Data	Data	Data
Däta	Däta	Däta



Consider: select * from table where A = "v"

A	B	C
Metadata		
Data	Data	Data
...	...	...
Data	Data	Data
Metadata		
Data	Data	Data
...	...	...
Data	Data	Data
Metadata		
Data	Data	Data
Data	Data	Data



Consider: select * from table where A = "v"

Can scan in $O(N/B)$ I/O

But this is wasteful if there are few matching rows

A	B	C
Metadata		
Data	Data	Data
...	...	...
Data	Data	Data
Metadata		
Data	Data	Data
...	...	...
Data	Data	Data
Metadata		
Data	Data	Data
Data	Data	Data



Consider: select * from table where A = "v"

Can scan in $O(N/B)$ I/O

But this is wasteful if there are few matching rows

Can we do better?

Techniques

(1) Zone Maps

(2) Sorted files

(3) Binary tree

(4) Hash table

(5) Extendible Hashing

(6) B-tree

(1) Zone Maps

A	...	min, max
61	...	
7	...	2, 61
2	...	
97	...	
90	...	32, 97
32	...	
32	...	
74	...	22, 74
22	...	

(1) Zone Maps

A	...
61	...
7	...
2	...
97	...
90	...
32	...
32	...
74	...
22	...

min, max
.....

2, 61

.....

32, 97

.....

22, 74

.....

Select * from table where **A = 10**

Skip

Skip

(1) Zone Maps

A	...	min, max
61	...	
7	...	2, 61
2	...	
97	...	
90	...	32, 97
32	...	
32	...	
74	...	22, 74
22	...	

Select * from table where **A = 10**

This still entails worst-case
 $O(N/B)$ I/Os

(1) Zone Maps

A	...	min, max
61	...	
7	...	2, 61
2	...	
97	...	
90	...	32, 97
32	...	
32	...	
74	...	22, 74
22	...	

Select * from table where **A = 10**

This still entails worst-case
 $O(N/B)$ I/Os

**This may be ok for analytics,
but it's not ok for many
applications, e.g., banking**

Techniques

(1) Zone Maps

(2) Sorted Files

(3) Binary Tree

(4) Hash Table

(5) Extendible Hashing

(6) B-tree

(2) Sorted Files

A	B	C
61	b1	c1
7	b2	c2
2	b3	c3
97	b4	c4
90	b5	c5
32	b6	c6
32	b7	c7
74	b8	c8
22	b9	c9

Select * from table where **A = 10**

Sort based on a column of interest

(2) Sorted Files

A	B	C
2	b3	c3
7	b2	c2
22	b9	c9
32	b6	c6
32	b7	c7
61	b1	c1
74	b8	c8
90	b5	c5
97	b4	c4

Select * from table where **A = 10**

Search algorithm?

(2) Sorted Files

A	B	C
2	b3	c3
7	b2	c2
.....		
22	b9	c9
32	b6	c6
.....		
32	b7	c7
61	b1	c1
.....		
74	b8	c8
90	b5	c5
.....		
97	b4	c4

Select * from table where **A = 10**

Search algorithm?

Binary search looking at first key in each block

Cost?

(2) Sorted Files

A	B	C
2	b3	c3
7	b2	c2
22	b9	c9
32	b6	c6
32	b7	c7
61	b1	c1
74	b8	c8
90	b5	c5
97	b4	c4

I/O Cost: $O(\log_2(N/B))$ I/O

Since we search over N/B blocks

(2) Sorted Files

A	B	C
2	b3	c3
7	b2	c2
22	b9	c9
32	b6	c6
32	b7	c7
61	b1	c1
74	b8	c8
90	b5	c5
97	b4	c4

I/O Cost: $O(\log_2(N/B))$ I/O

Since we search over N/B blocks

CPU Cost?

(2) Sorted Files

A	B	C
2	b3	c3
7	b2	c2
22	b9	c9
32	b6	c6
32	b7	c7
61	b1	c1
74	b8	c8
90	b5	c5
97	b4	c4

I/O Cost: $O(\log_2(N/B))$ I/O

Since we search over N/B blocks

**CPU Cost: $O(\log_2(N/B) + \log_2(B))$
 $= O(\log_2(N))$**

(2) Sorted Files

A	B	C
2	b3	c3
7	b2	c2
22	b9	c9
32	b6	c6
32	b7	c7
61	b1	c1
74	b8	c8
90	b5	c5
97	b4	c4

Update/insert algorithm?

e.g., insert A = 50

(2) Sorted Files

A	B	C
2	b3	c3
7	b2	c2
22	b9	c9
32	b6	c6
32	b7	c7
50	b10	c10
61	b1	c1
74	b8	c8
90	b5	c5
97	b4	c4

Update/insert algorithm?

Binary search to find proper location, and then push rest of elements up by one slot

Cost?

(2) Sorted Files

A	B	C
2	b3	c3
7	b2	c2
22	b9	c9
32	b6	c6
32	b7	c7
50	b10	c10
61	b1	c1
74	b8	c8
90	b5	c5
97	b4	c4

Update/insert algorithm?

Binary search to find proper location, and then push rest of elements up by one slot

Cost? $O(N/B)$

Expensive!

(2) Sorted Files

A	B	C
2	b3	c3
7	b2	c2
22	b9	c9
32	b6	c6
32	b7	c7
50	b10	c10
61	b1	c1
74	b8	c8
90	b5	c5
97	b4	c4

Can you spot another disadvantage?

(2) Sorted Files

A	B	C
2	b3	c3
7	b2	c2
22	b9	c9
32	b6	c6
32	b7	c7
50	b10	c10
61	b1	c1
74	b8	c8
90	b5	c5
97	b4	c4

Can you spot another disadvantage?

Can only sort based on one column, but what if we want to search across other columns?

e.g., Select * from table where **B** = "..."

Techniques

(1) Zone Maps

(2) Sorted Files

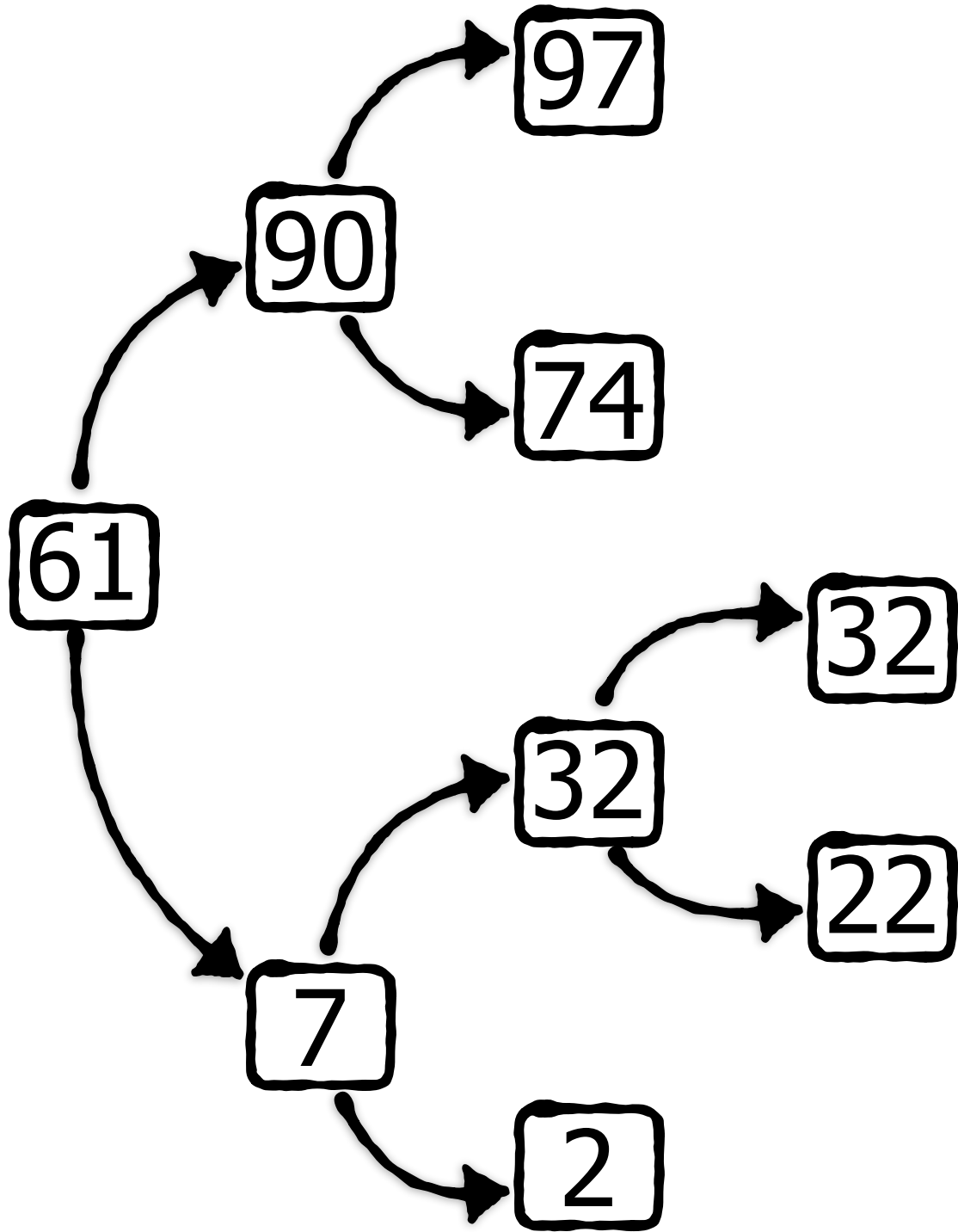
(3) Binary Tree

(4) Hash table

(5) Extendible Hashing

(6) B-tree

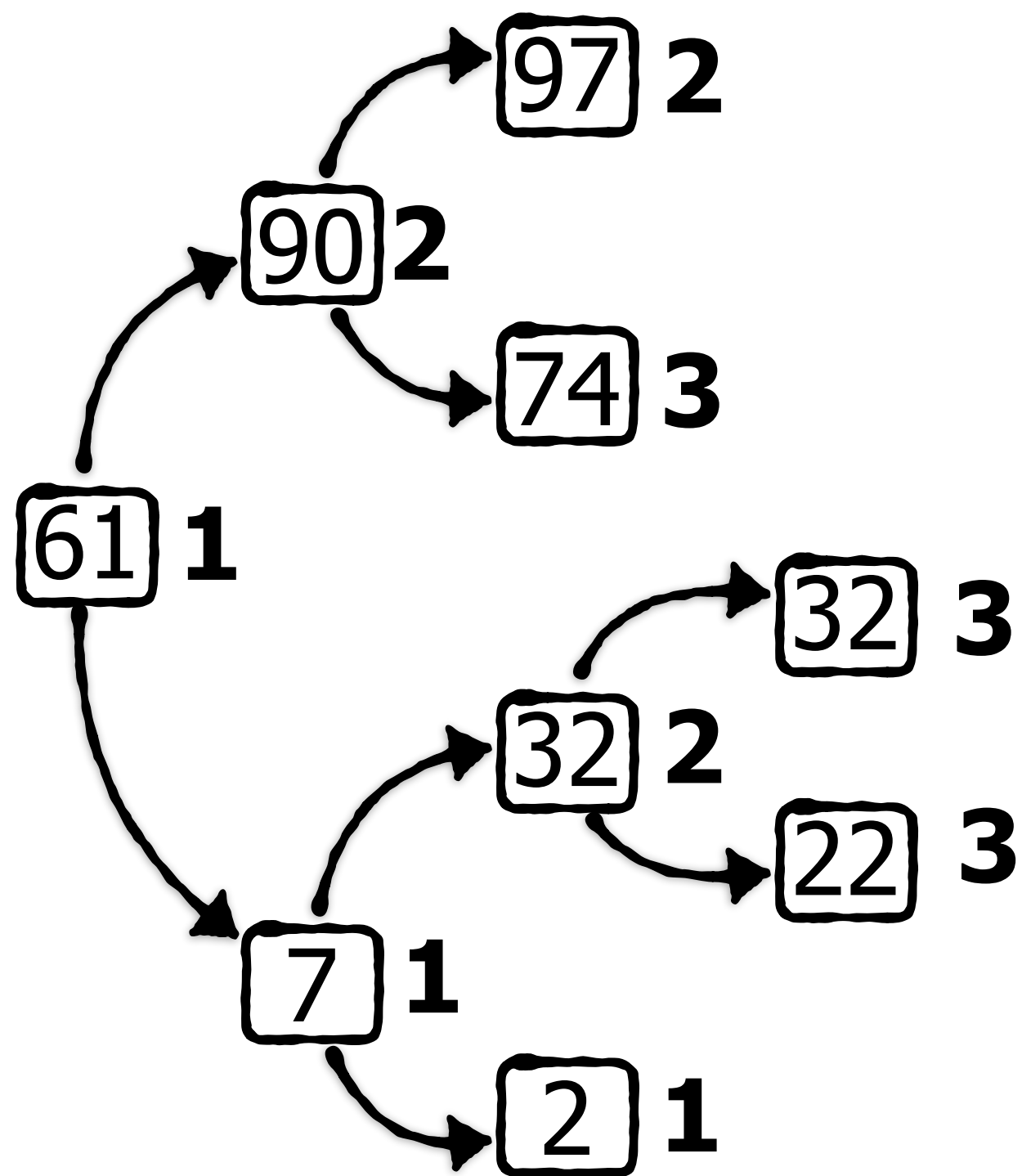
(3) Binary Tree



	A	B	C
1	61	b1	c1
1	7	b2	c2
1	2	b3	c3
2	97	b4	c4
2	90	b5	c5
2	32	b6	c6
3	32	b7	c7
3	74	b8	c8
3	22	b9	c9

Use a binary tree to map each key to the page it resides in

(3) Binary Tree



A	B	C
61	b1	c1
7	b2	c2
2	b3	c3
...	...	...
97	b4	c4
90	b5	c5
32	b6	c6
...	...	...
32	b7	c7
74	b8	c8
22	b9	c9

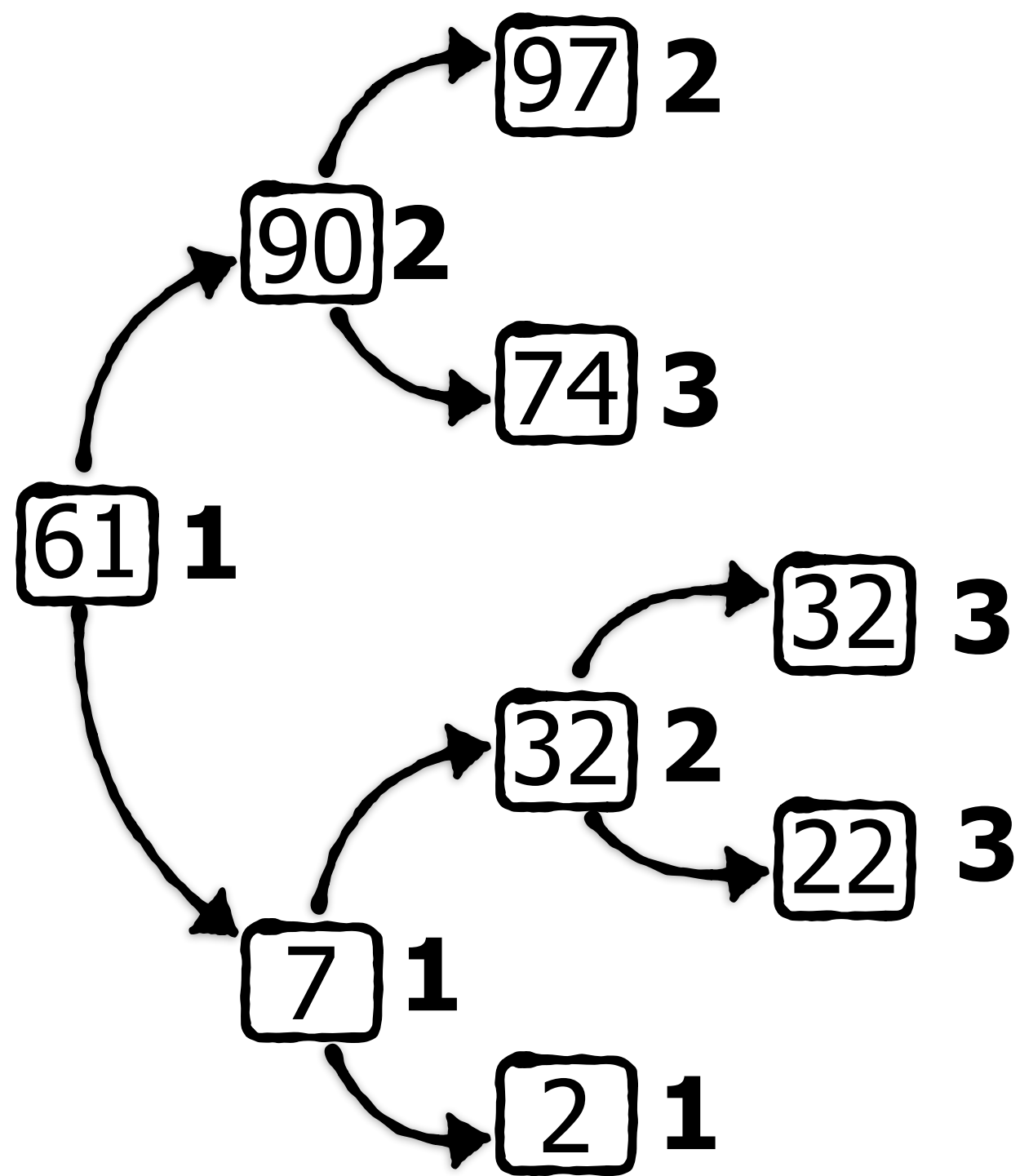
Use a binary tree to map each key to the page it resides in

Tree must be stored in storage due to its size and for persistence

(3) Binary Tree

I/O cost:

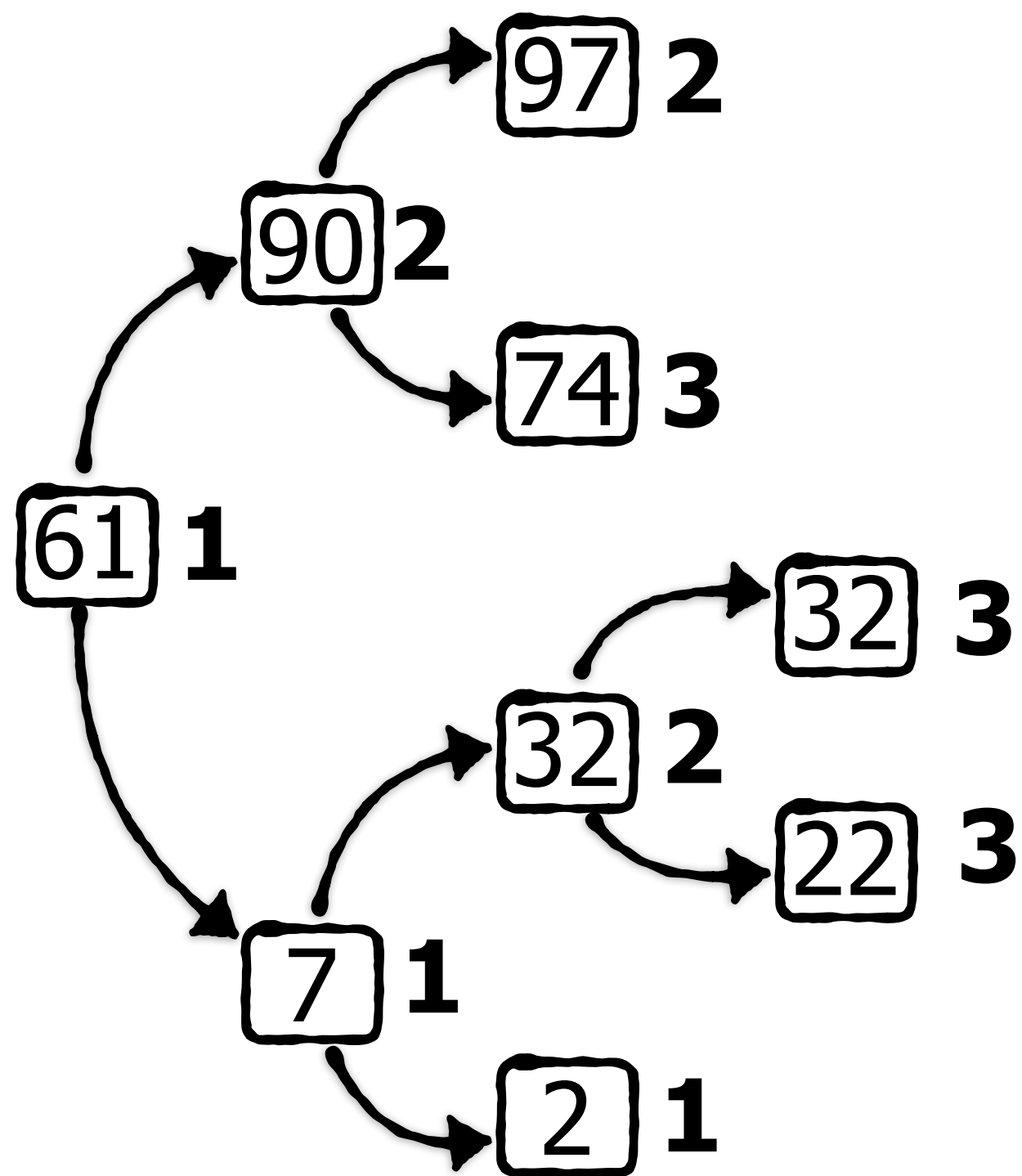
CPU cost:



(3) Binary Tree

I/O cost: $O(\log_2 N)$

CPU cost: $O(\log_2 N)$



Techniques

(1) Zone Maps

(2) Sorted Files

(3) Binary Tree

(4) Hash Table

(5) Extendible Hashing

(6) B-tree

(4) Hash Table - in storage

(4) Hash Table - in storage

Table

	A	...
0	61	...
	7	...
	2	...
.....		
1	97	...
	90	...
	32	...
.....		
2	32	...
	74	...
	22	...

(4) Hash Table - in storage

Each bucket contains B entries mapping a key to its data page

Hash table
 $\text{hash}(x) = x \% 3$

0	90 → 1	
		
1	61 → 0	97 → 1
	7 → 0	22 → 2
		
2	2 → 0	32 → 2
	32 → 1	74 → 2

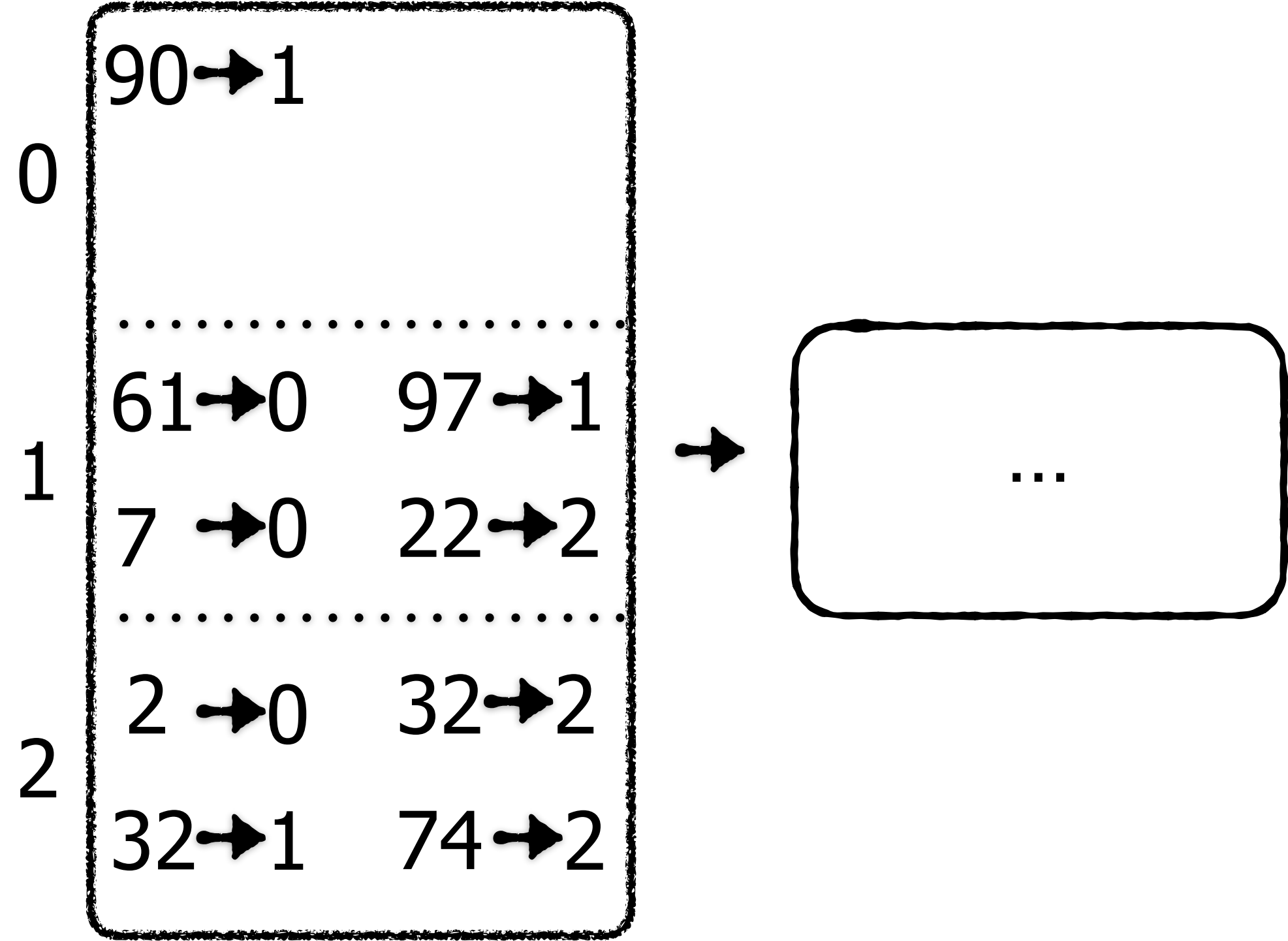
Table

A		...
0	61	...
	7	...
	2	...
1	97	...
	90	...
	32	...
2	32	...
	74	...
	22	...

(4) Hash Table - in storage

Collision resolution:
chaining, probing, etc.

Hash table
 $\text{hash}(x) = x \% 3$



(4) Hash Table - in storage

Double when reaching capacity.

$$\text{hash}(x) = x \% 3$$

0	90 → 1	
		
1	61 → 0 97 → 1	
	7 → 0 22 → 2	
		
2	2 → 0 32 → 2	
	32 → 1 74 → 2	

Rehash



$$\text{hash}(x) = x \% 6$$

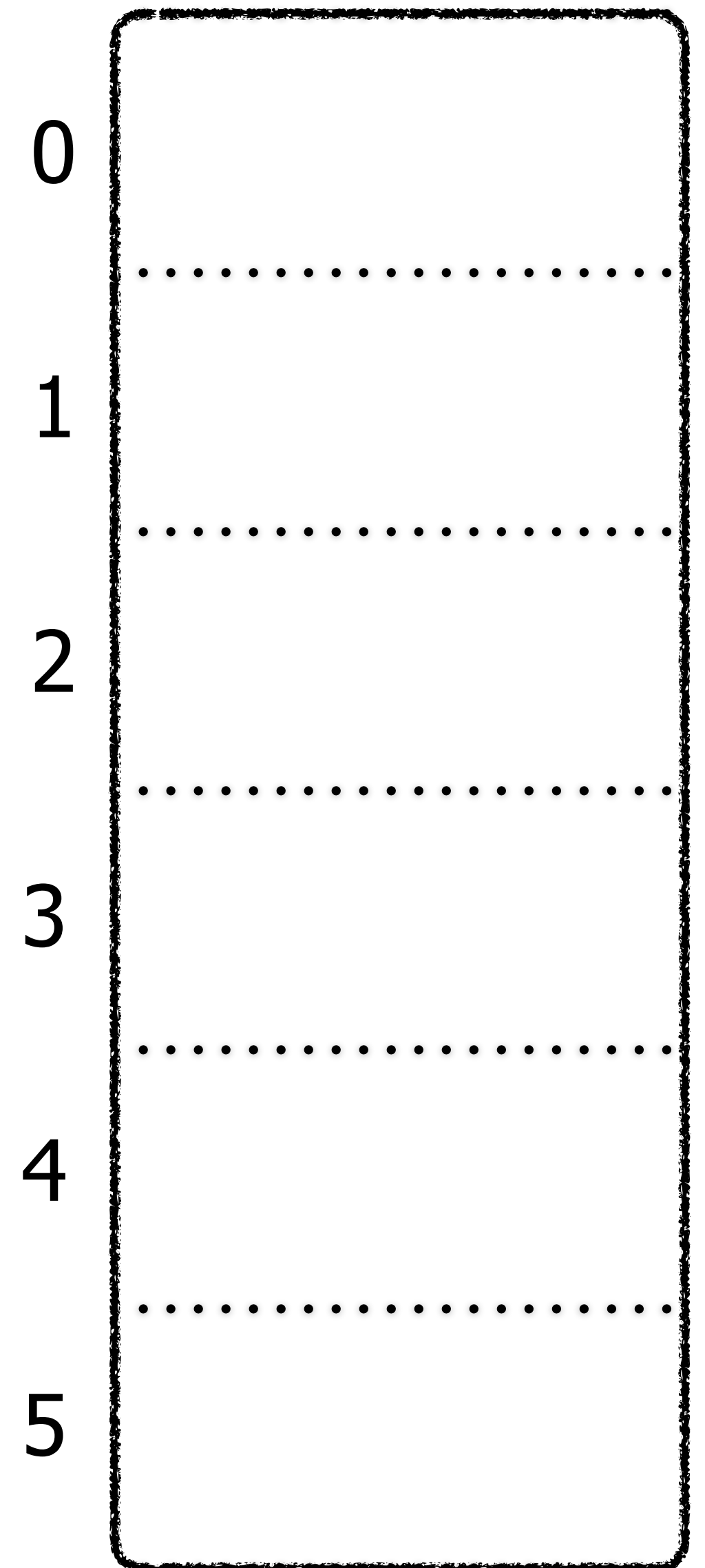
0	
	
1	
	
2	
	
3	
	
4	
	
5	

(4) Hash Table - in storage

Read/write I/O cost:

$O(1)$ per random query assuming a good hash function and no key repetitions

$$\text{hash}(x) = x \% 6$$



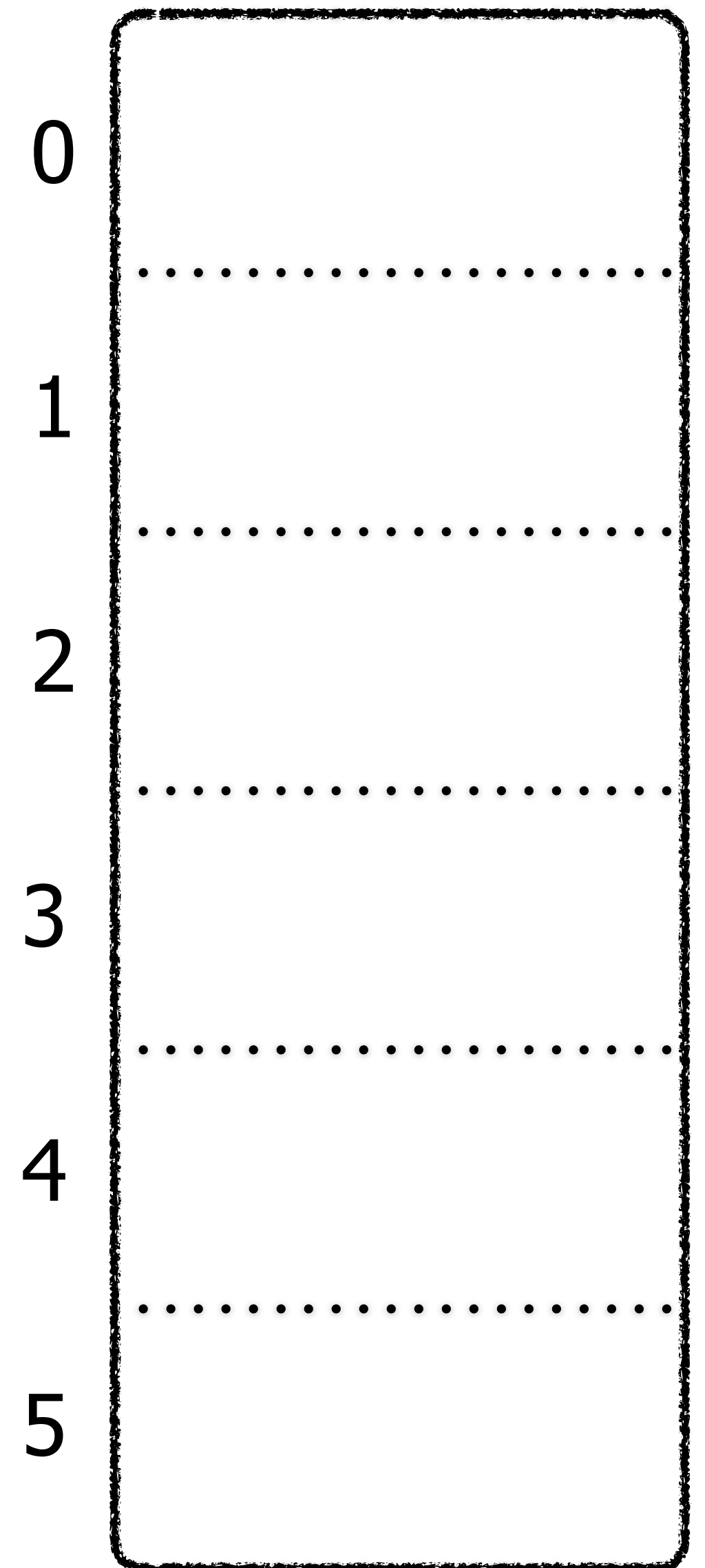
(4) Hash Table - in storage

Read/write I/O cost:

$O(1)$ per random query assuming a good hash function and no key repetitions

Downsides?

$$\text{hash}(x) = x \% 6$$



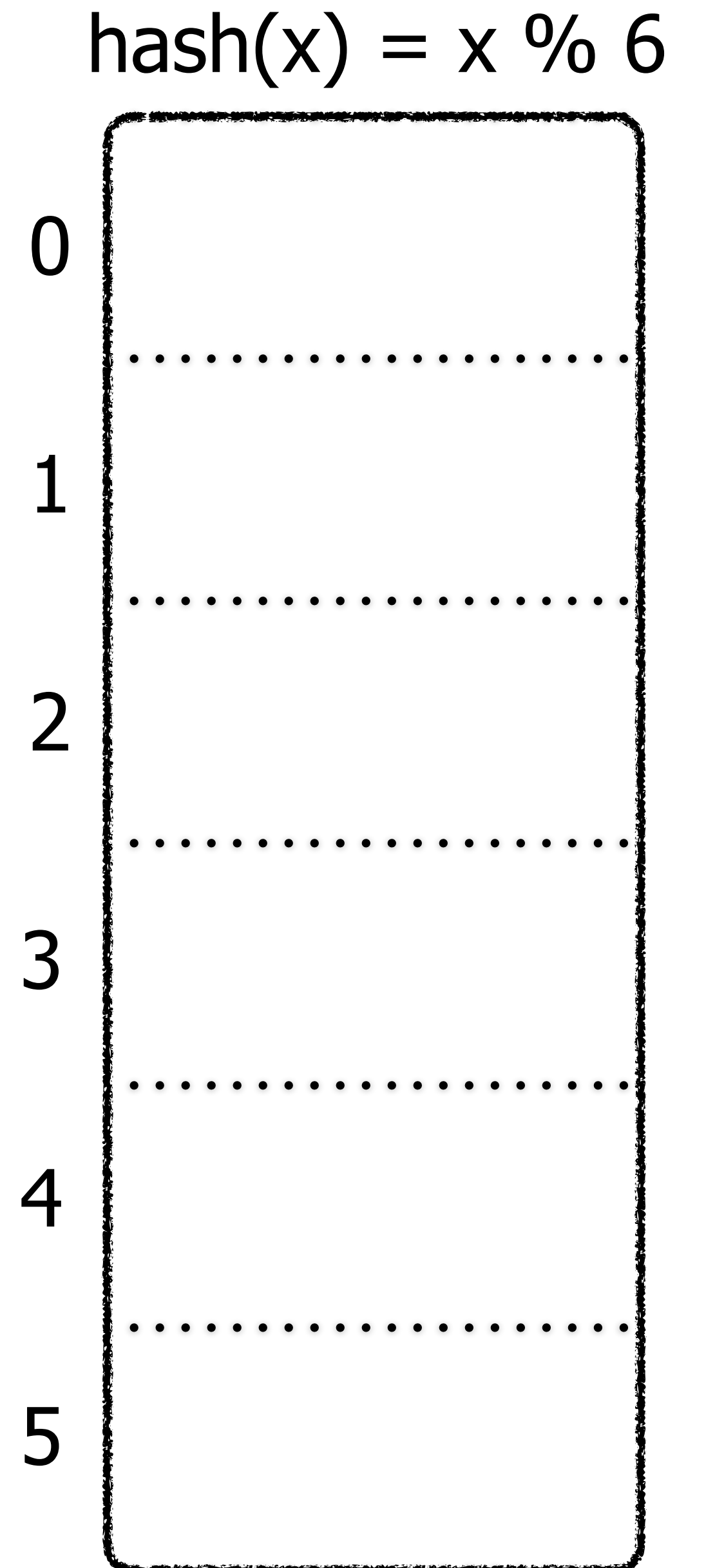
(4) Hash Table - in storage

Read/write I/O cost:

$O(1)$ per random query assuming a good hash function and no key repetitions

Downsides?

- (1) No support for range reads
- (2) Expansion leads to performance slumps
- (3) We waste 50% of capacity right after expansion



(4) Hash Table - in storage

Read/write I/O cost:

$O(1)$ per random query assuming a good hash function and no key repetitions

Downsides?

(1) No support for range reads

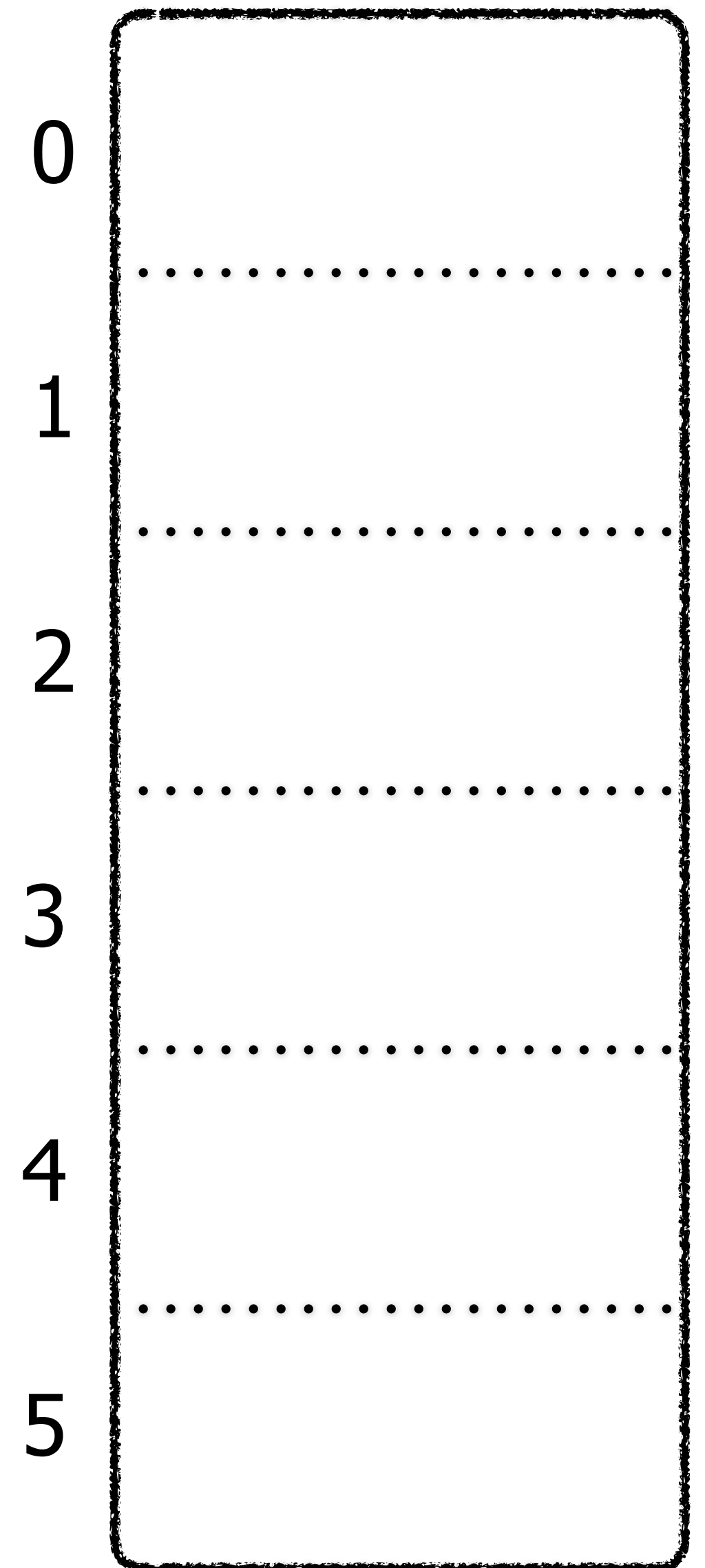
(2) Expansion leads to performance slumps

(3) We waste 50% of capacity right after expansion



Let's address points (2) and (3).

$$\text{hash}(x) = x \% 6$$



(4) Hash Table - in storage

Read/write I/O cost:

$O(1)$ per random query assuming a good hash function and no key repetitions

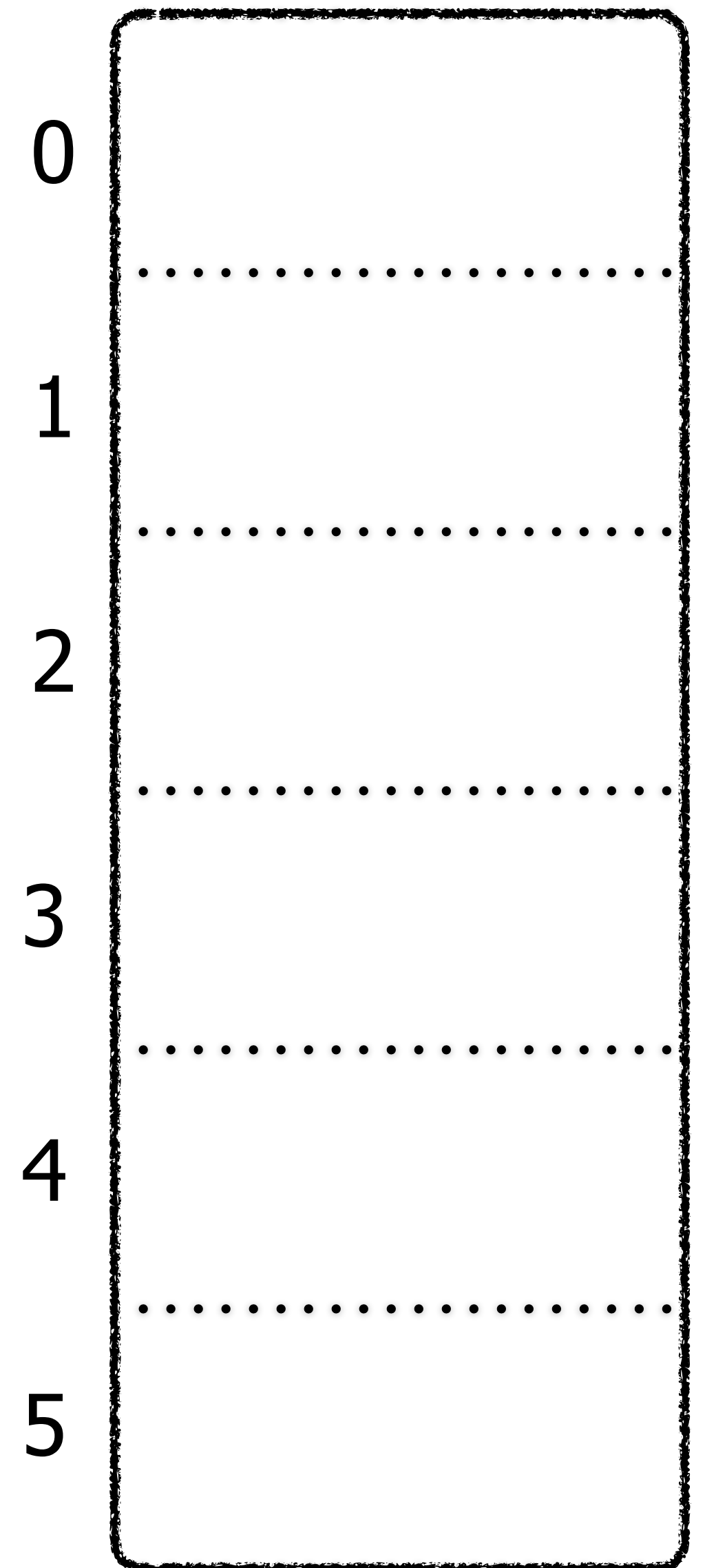
Downsides?

- (1) No support for range reads
- (2) Expansion leads to performance slumps
- (3) We waste 50% of capacity right after expansion



Let's address points (2) and (3). **Any ideas?**

$$\text{hash}(x) = x \% 6$$



Techniques

(1) Zone Maps

(2) Sorted Files

(3) Binary Tree

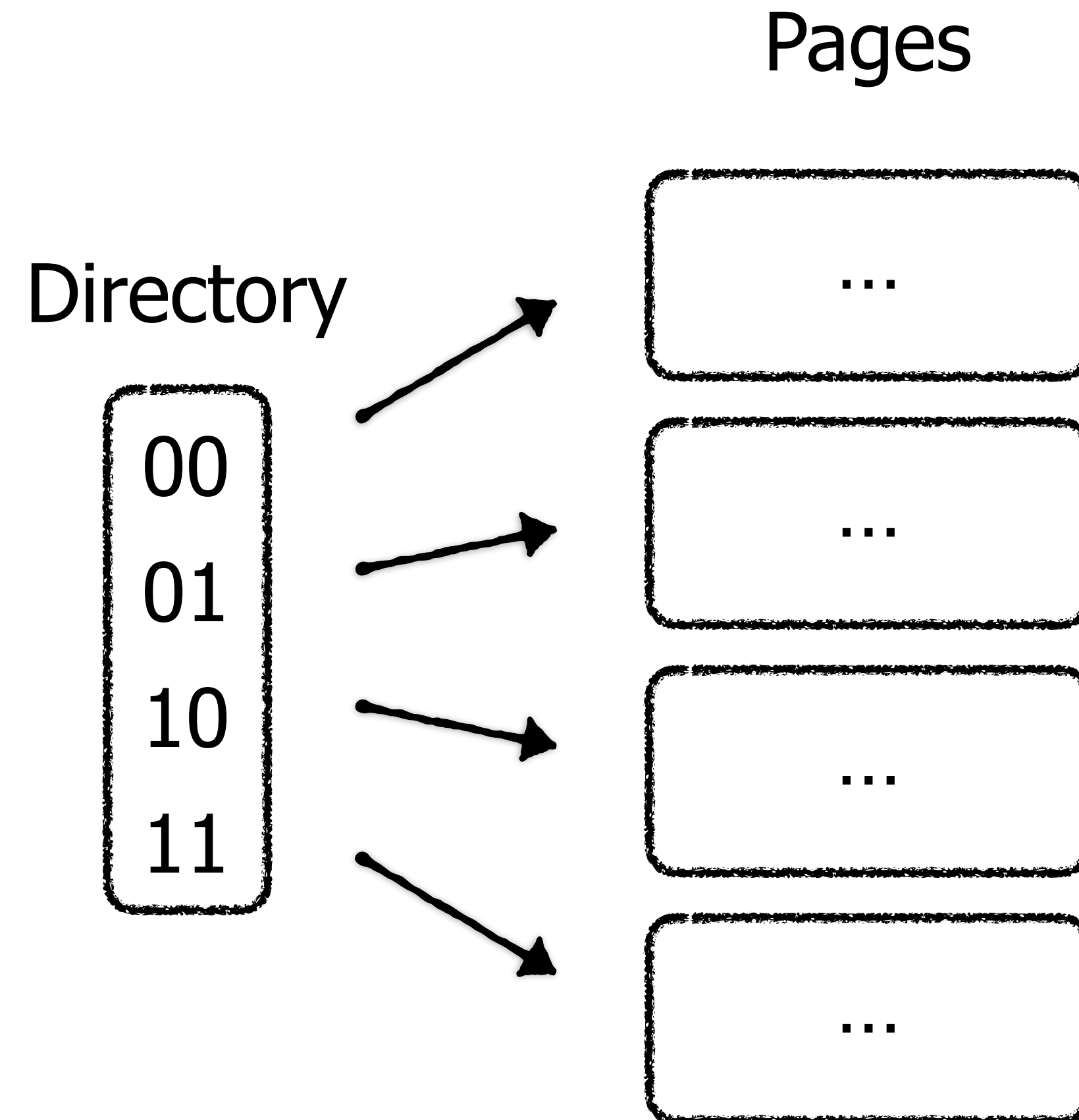
(4) Hash Table

(5) Extendible Hashing

(6) B-tree

(5) Extendible Hashing

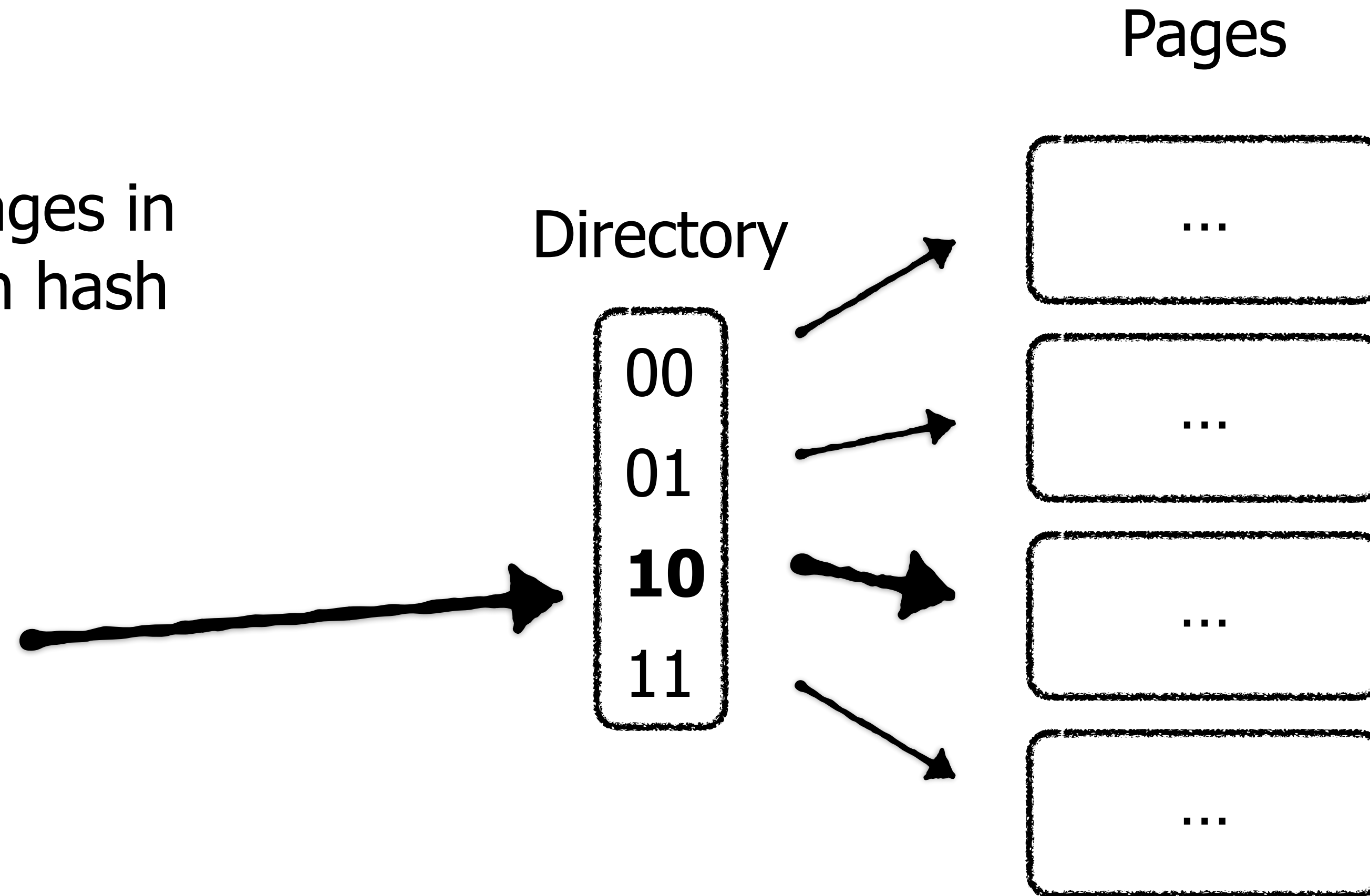
A directory maps pages in storage with a given hash suffix



(5) Extendible Hashing

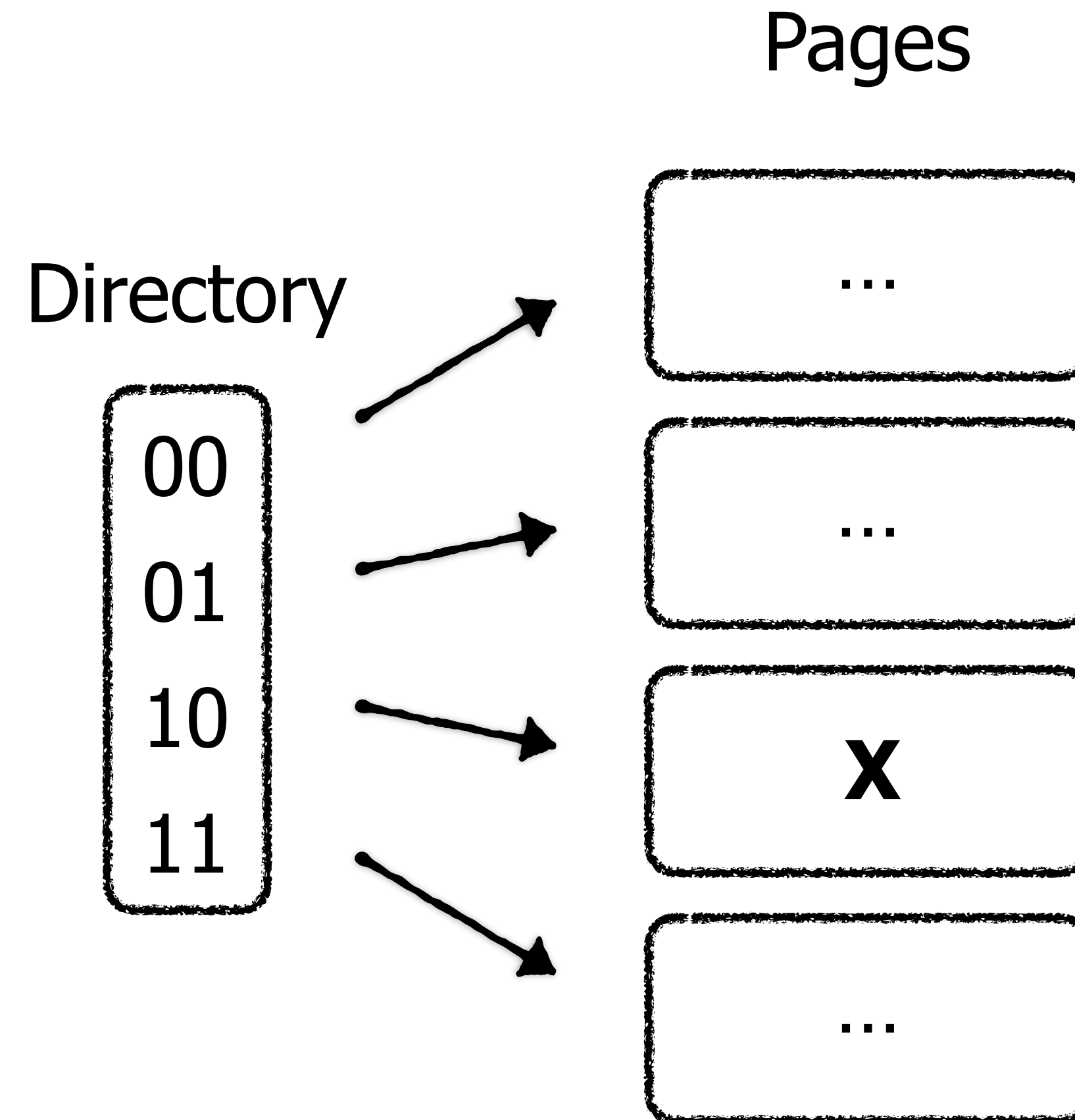
A directory maps pages in storage with a given hash suffix

e.g., insert key X
 $\text{Hash}(X) = \dots 10$



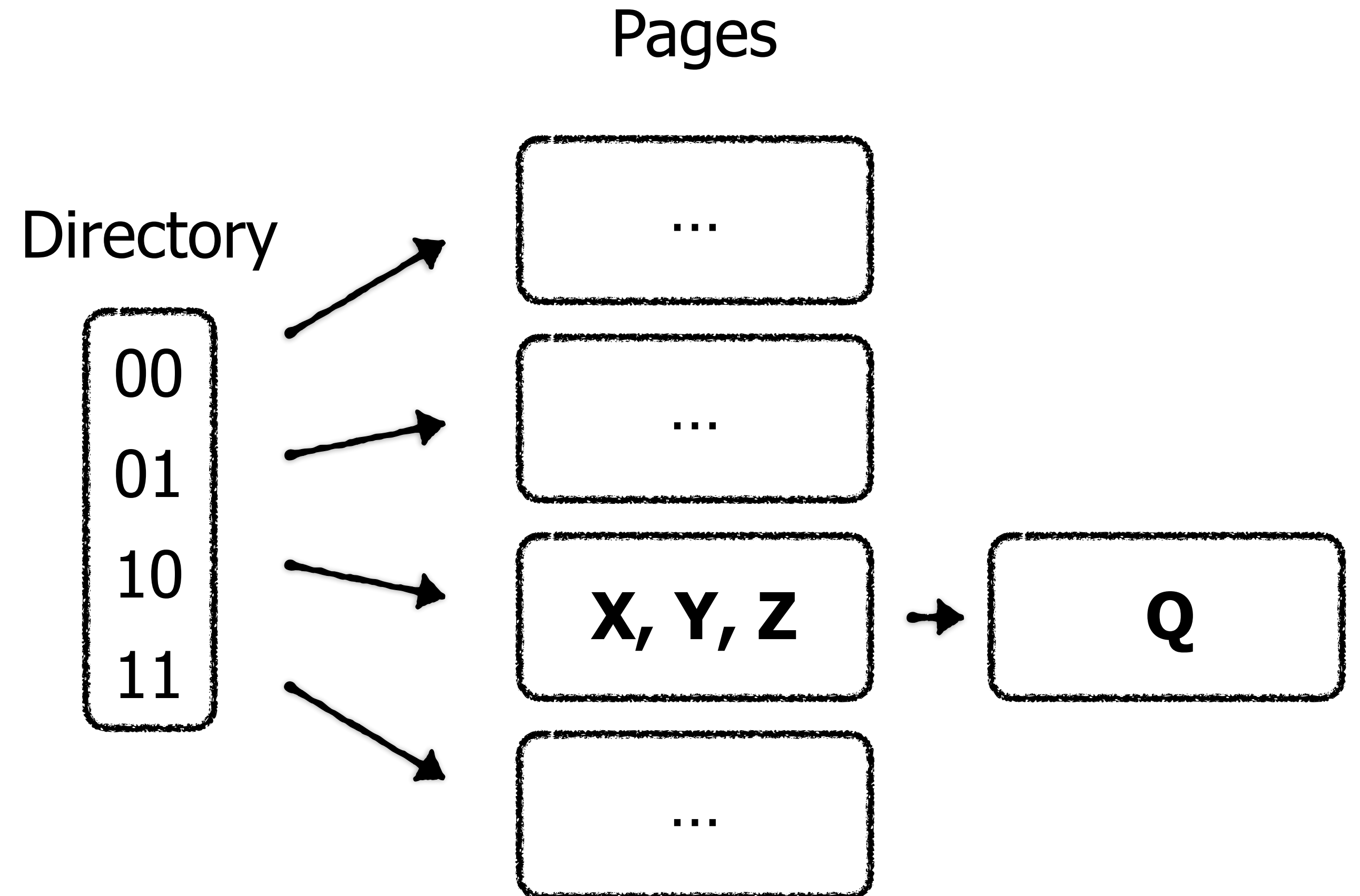
(5) Extendible Hashing

A directory maps pages in storage with a given hash suffix



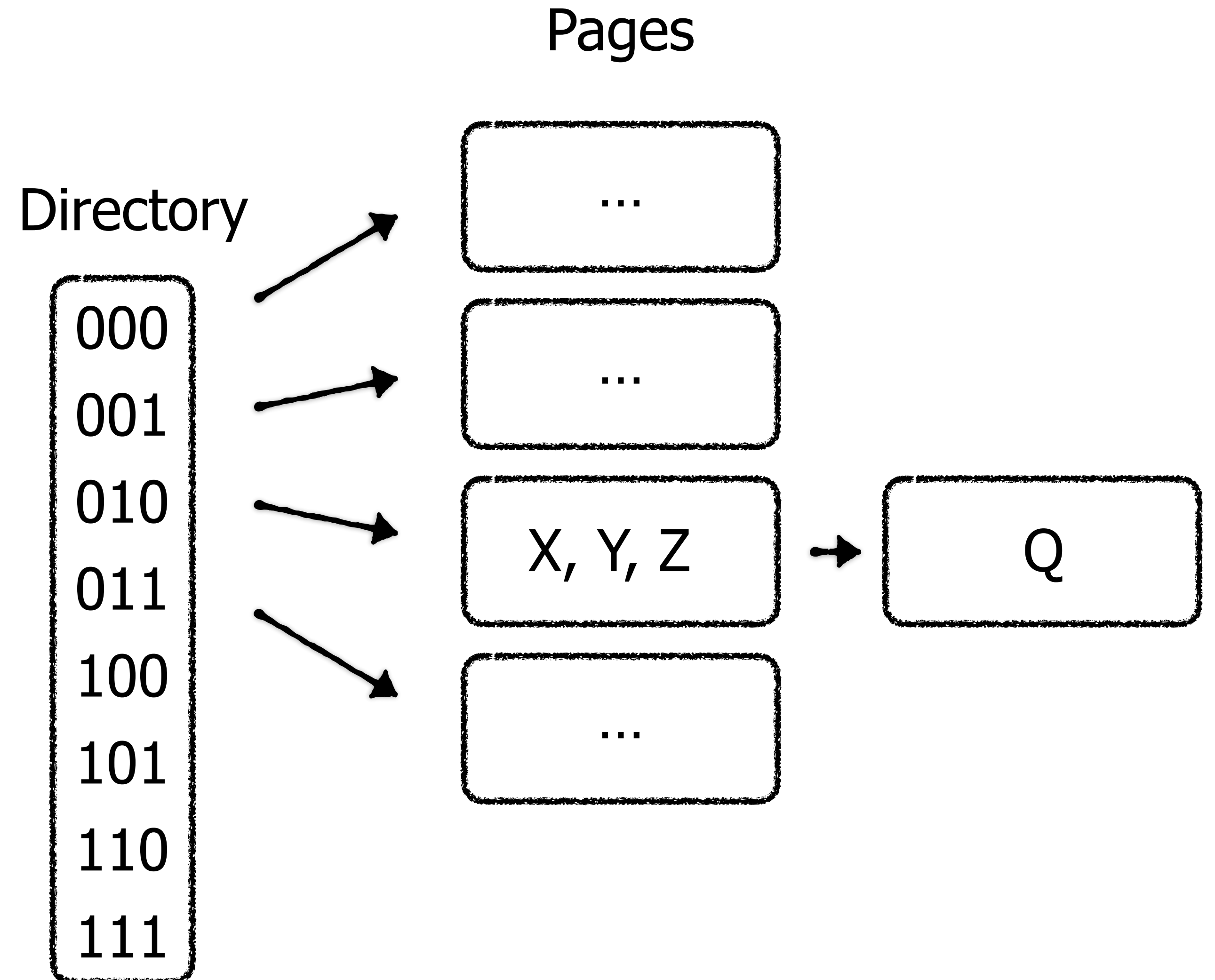
(5) Extendible Hashing

Handle overflows via chaining



(5) Extendible Hashing

When we reach capacity,
double directory size.



(5) Extendible Hashing

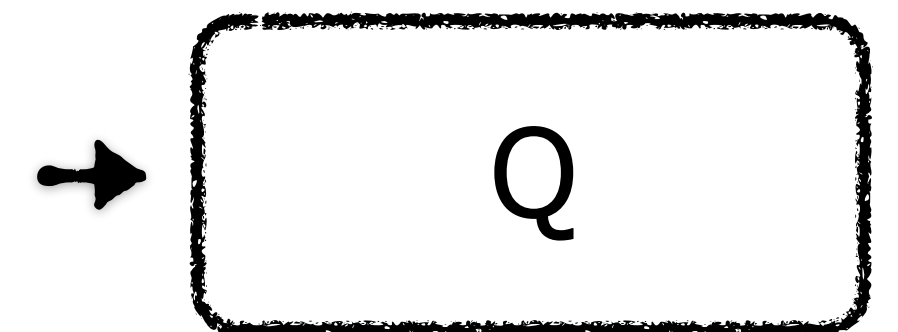
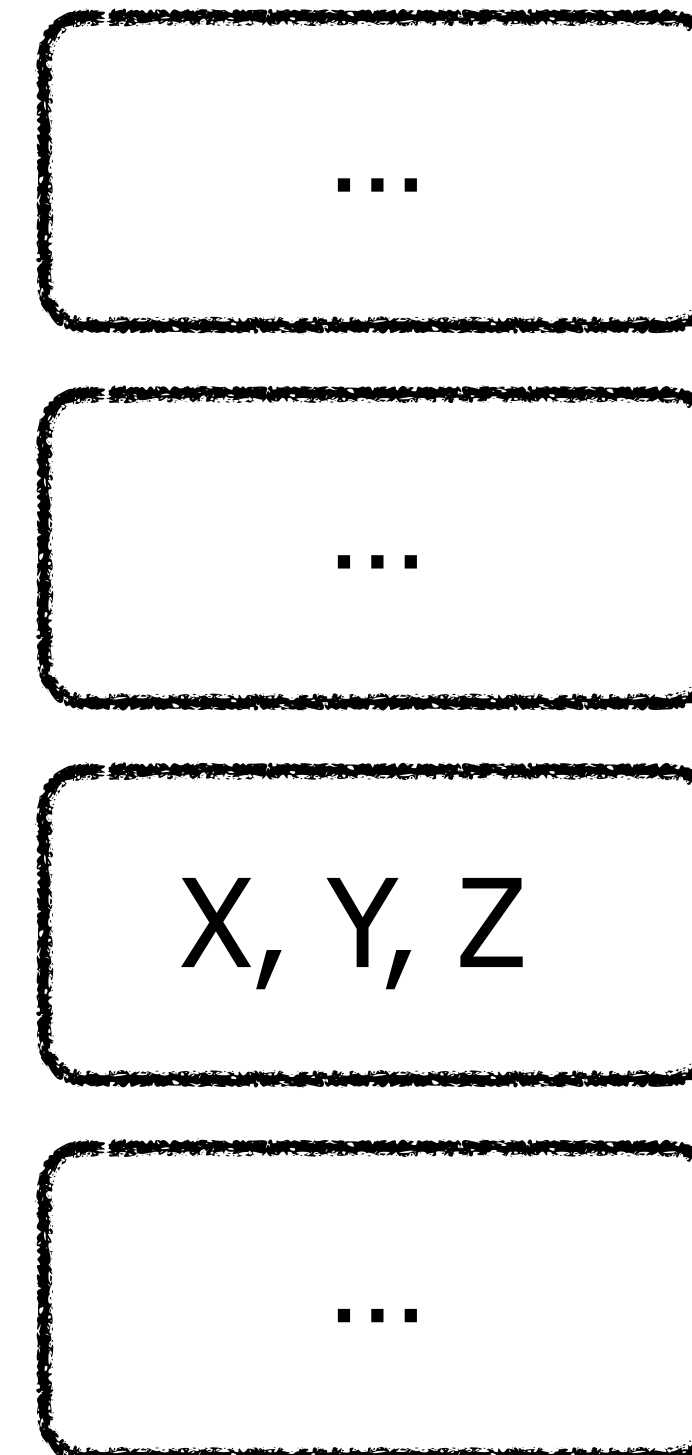
When we reach capacity,
double directory size.

New directory slots still point to
previous pages

Directory

000
001
010
011
100
101
110
111

Pages



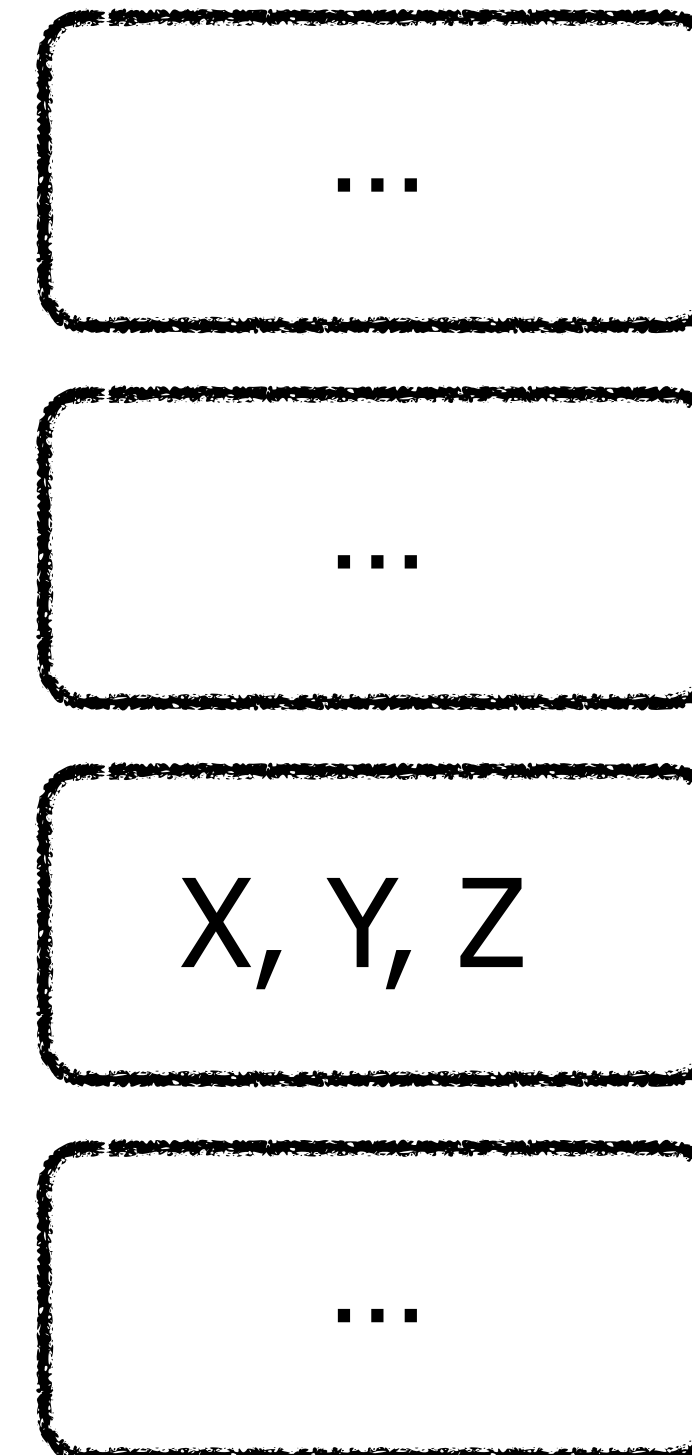
(5) Extendible Hashing

We can now expand one overflowing bucket at a time.

Directory

000
001
010
011
100
101
110
111

Pages



(5) Extendible Hashing

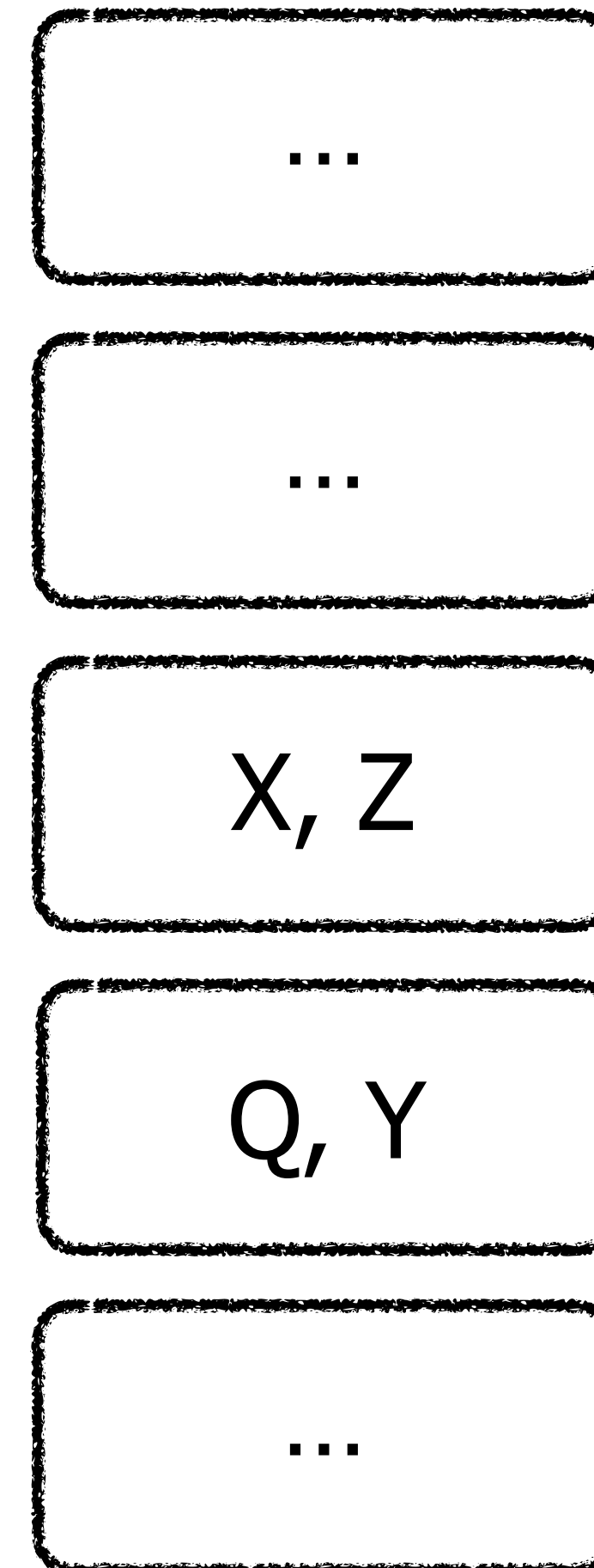
We can now expand one overflowing bucket at a time.

Rehash entries to decide which of two buckets they should go into

Directory

000
001
010
011
100
101
110
111

Pages



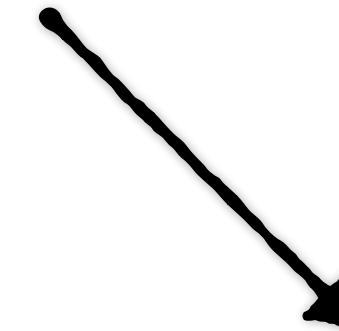
(5) Extendible Hashing

Comparison to Hash Table?

Directory

Pages

000
001
010
011
100
101
110
111



...

...

X, Z

Q, Y

...

(5) Extendible Hashing

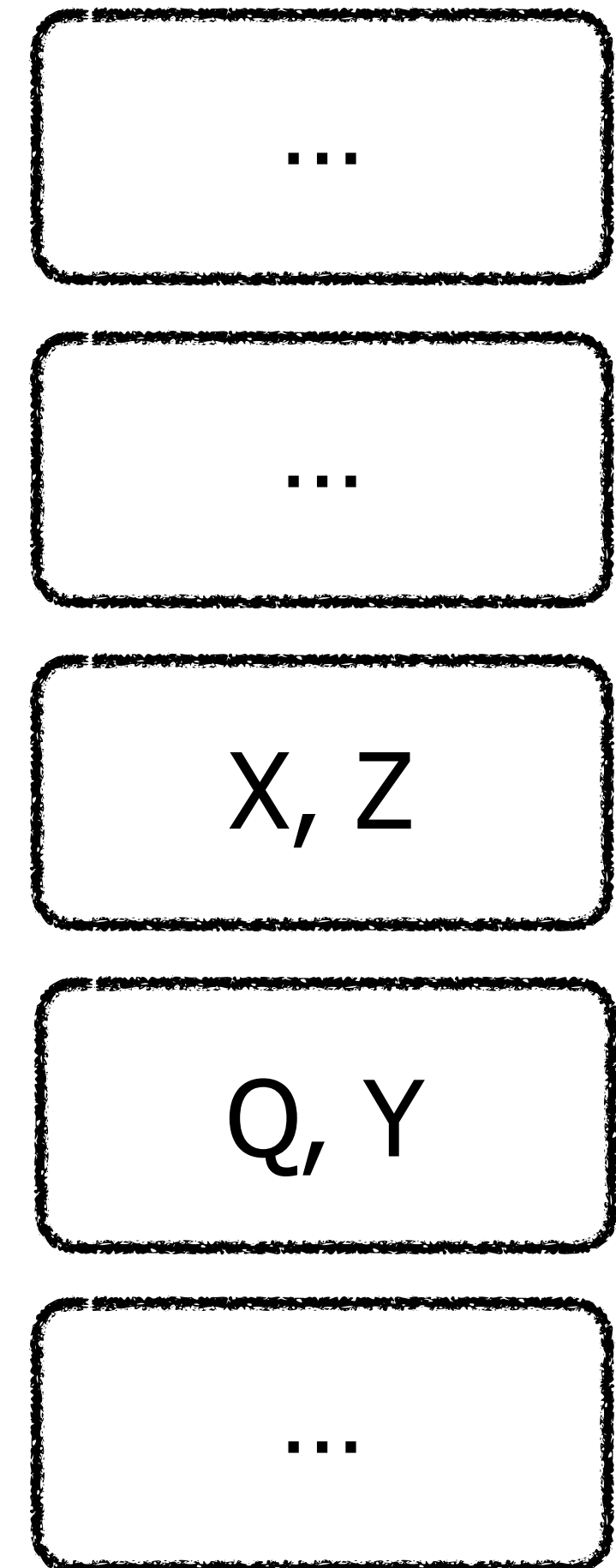
Comparison to Hash Table

- (1) Less performance slumps due to expansion
- (2) Avoid wasting 50% space after expansion

Directory

000
001
010
011
100
101
110
111

Pages



(5) Extendible Hashing

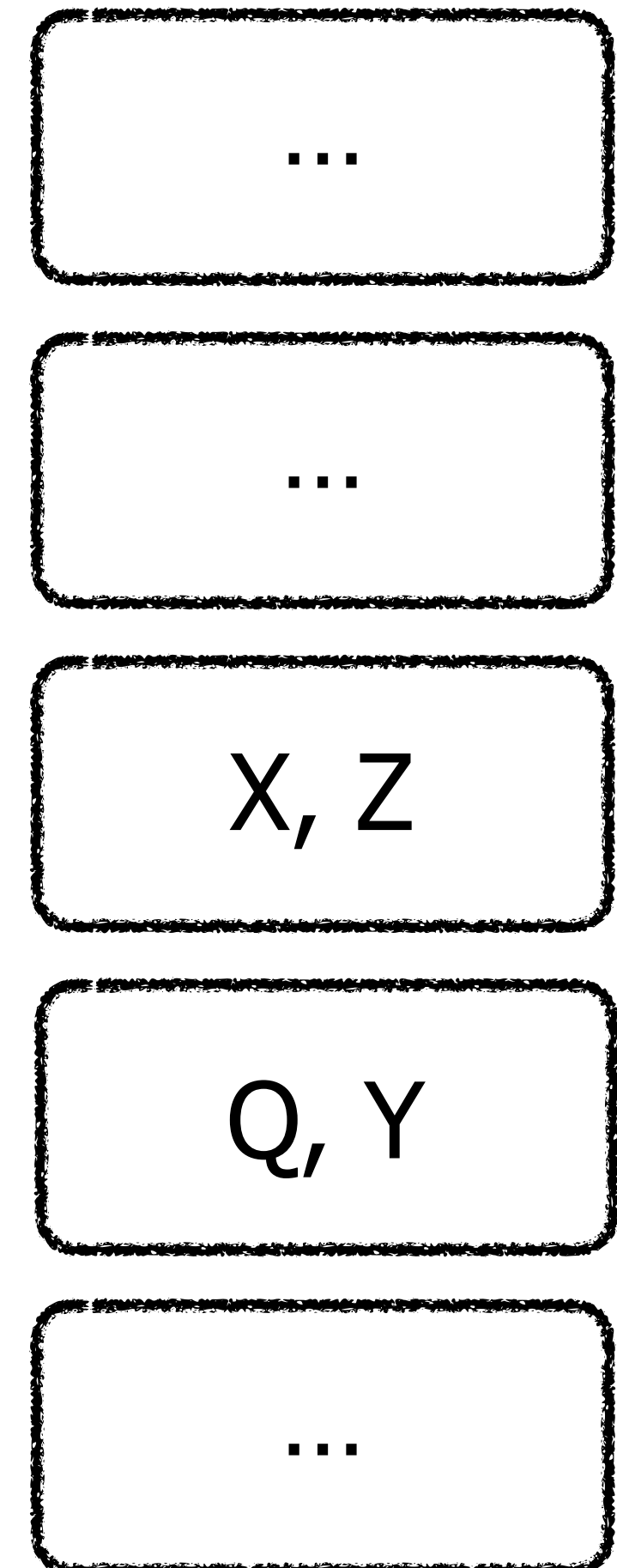
Comparison to Hash Table

- (1) Less performance slumps due to expansion
- (2) Avoid wasting 50% space after expansion
- (3) Directory consumes memory if it is cached, or it requires one extra I/O per operation if in storage.**

Directory

000
001
010
011
100
101
110
111

Pages



(5) Extendible Hashing

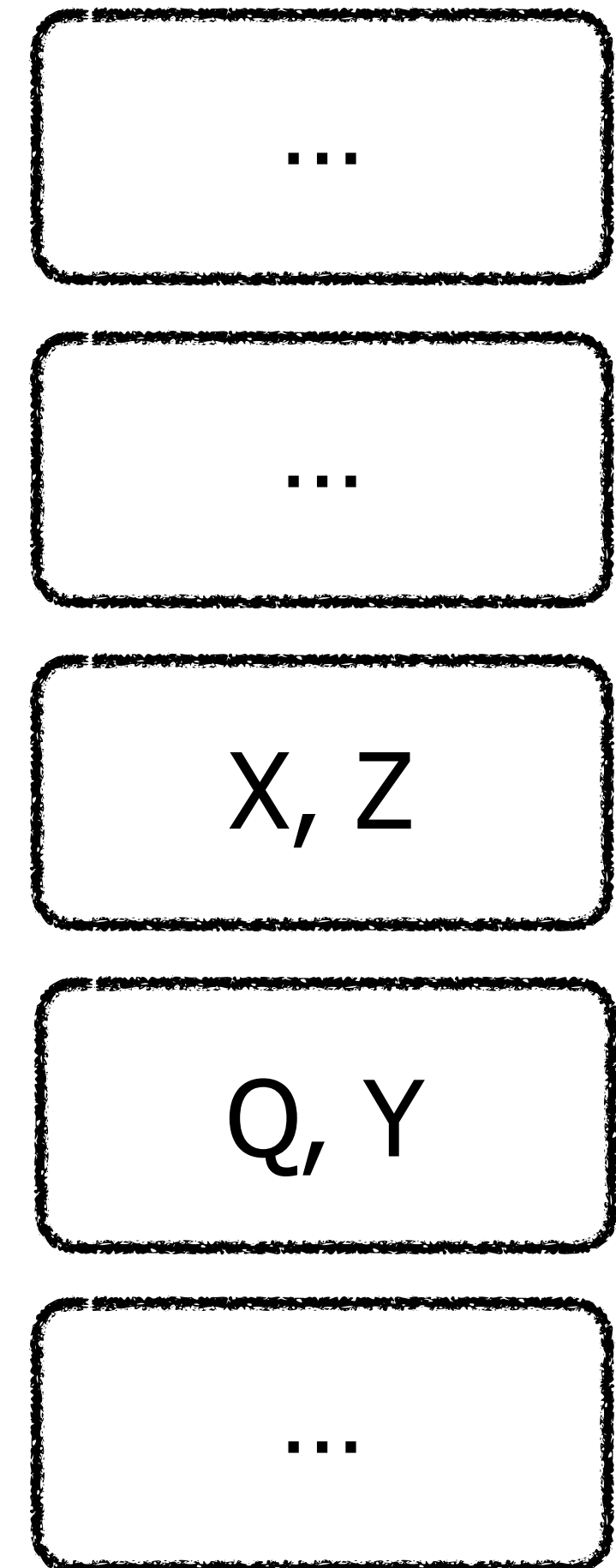
Comparison to Hash Table

- (1) Less performance slumps due to expansion
- (2) Avoid wasting 50% space after expansion
- (3) Directory consumes memory if it is cached, or it requires one extra I/O per operation if in storage.

Directory

000
001
010
011
100
101
110
111

Pages



This is nice, but it can't support range queries.

Techniques

(1) Zone Maps

(2) Sorted Files

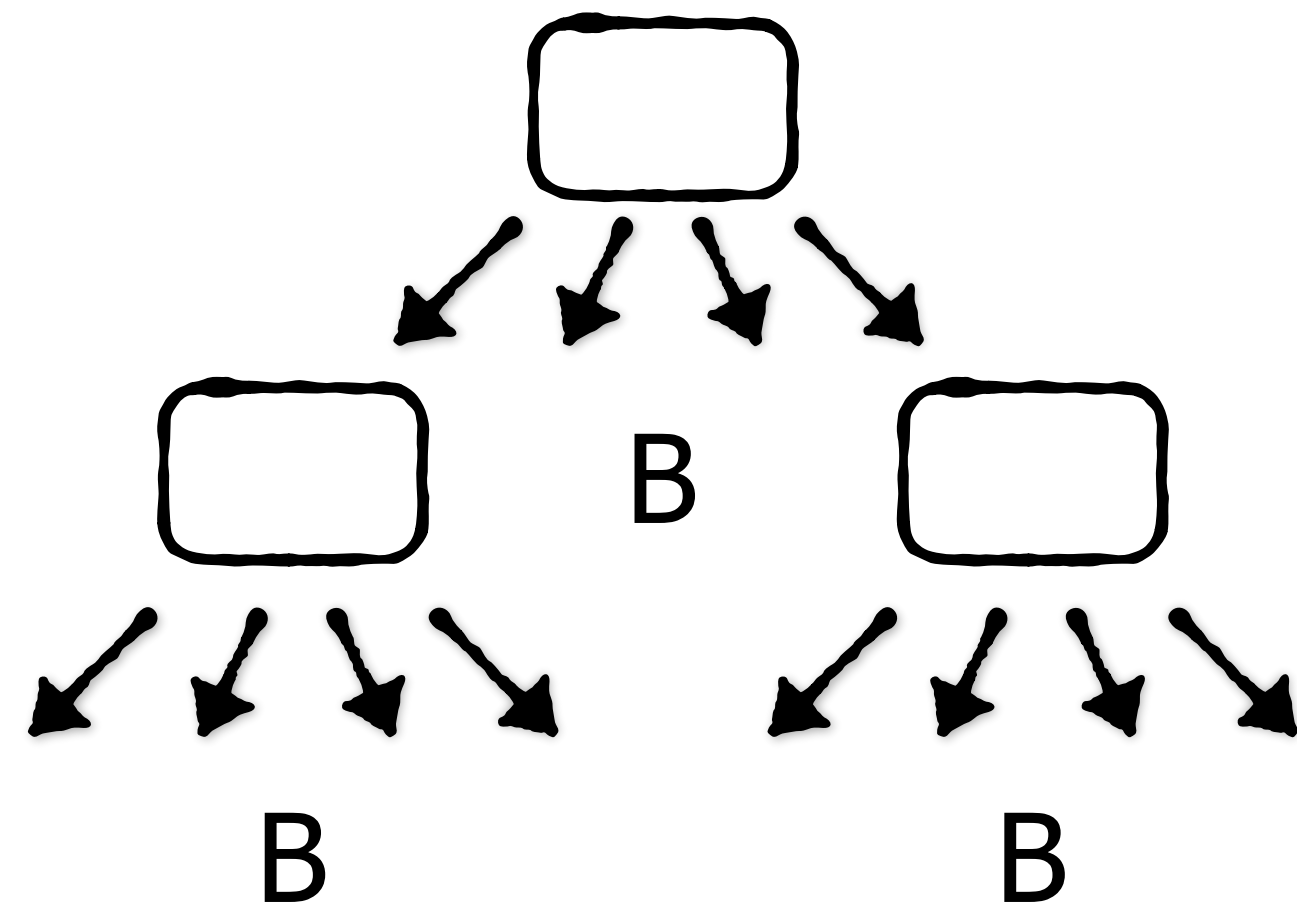
(3) Binary Tree

(4) Hash Table

(5) Extendible Hashing

(6) B-Tree

(6) B-Tree



Invented in 1970 by Rudolph Bayer

Used ubiquitously in database systems since then

Core idea: since we have to read at least B entries from storage at a time, let's make each tree node have B entries to prune the search space by a factor of B .

(6) B-Tree

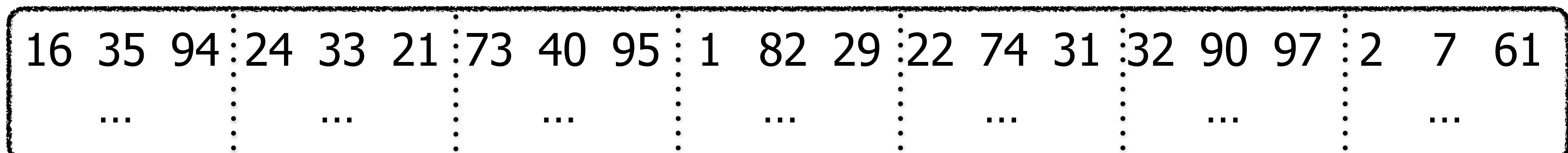
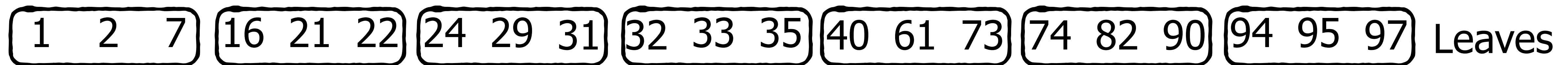
In this example, let's assume $B=3$, meaning each node contains 3 keys.

16	35	94	24	33	21	73	40	95	1	82	29	22	74	31	32	90	97	2	7	61
...			...			...			...			...			...			...		

Table
Col A

(6) B-Tree

In this example, let's assume $B=3$, meaning each node contains 3 keys.



(6) B-Tree

In this example, let's assume $B=3$, meaning each node contains 3 keys.

From each key, point to page the entry resides in

Diagram illustrating the mapping of 7 pages to 21 slots in a table:

Page #	Slot 1	Slot 2	Slot 3
7	1	2	7
6	16	21	22
5	24	29	31
4	32	33	35
3	40	61	73
2	74	82	90
1	94	95	97

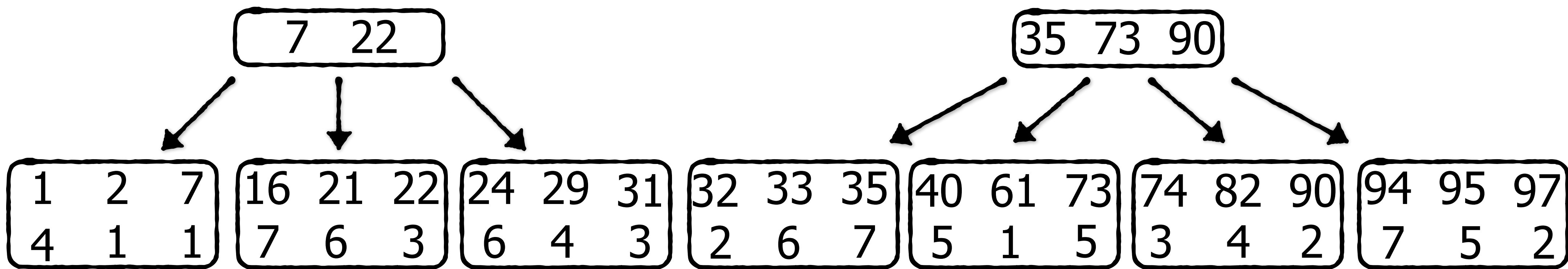
(6) B-Tree

In this example, let's assume $B=3$, meaning each node contains 3 keys.

From each key, point to page the entry resides in

Diagram illustrating a sequence of 21 numbers grouped into 7 boxes of 3. The boxes are labeled 7, 6, 5, 4, 3, 2, 1 from left to right. The numbers in the boxes are: Box 7: 1, 2, 7; Box 6: 16, 21, 22; Box 5: 24, 29, 31; Box 4: 32, 33, 35; Box 3: 40, 61, 73; Box 2: 74, 82, 90; Box 1: 94, 95, 97. Below the boxes, a table shows the sequence of numbers: 16, 35, 94, 24, 33, 21, 73, 40, 95, 1, 82, 29, 22, 74, 31, 32, 90, 97, 2, 7, 61. The table is labeled "Table" and "Page #".

The next layer is of internal nodes, containing a delimiting key to separate between each pair of adjacent leaves.

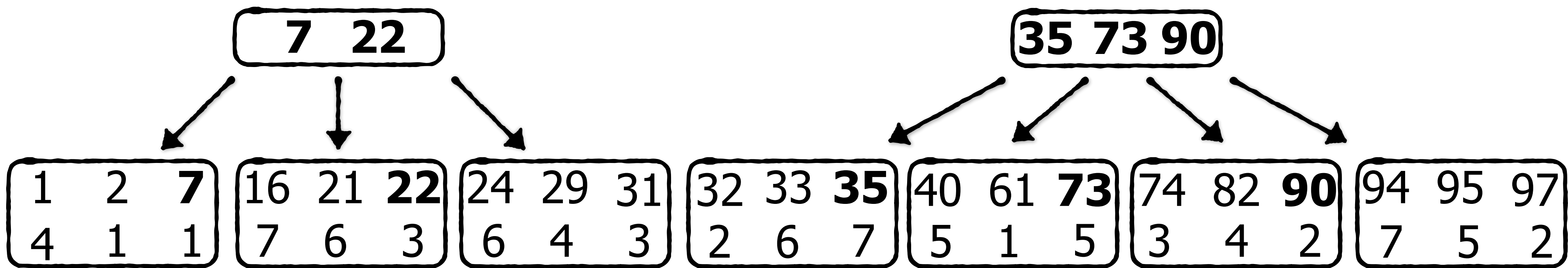


7			6			5			4			3			2			1		
16	35	94	24	33	21	73	40	95	1	82	29	22	74	31	32	90	97	2	7	61
...			...			...			...			...			...			...		

Page #
Table

The next layer is of internal nodes, containing a delimiting key to separate between each pair of adjacent leaves.

In this case, the delimiter is the largest key in sub-tree (except the last).

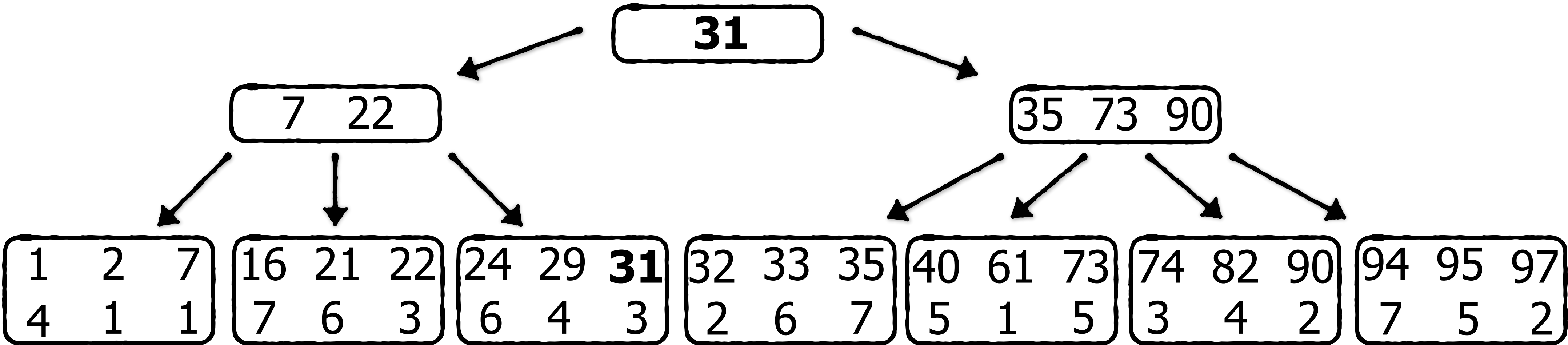


7			6			5			4			3			2			1		
16	35	94	24	33	21	73	40	95	1	82	29	22	74	31	32	90	97	2	7	61
...			...			...			...			...			...			...		

Page #
Table

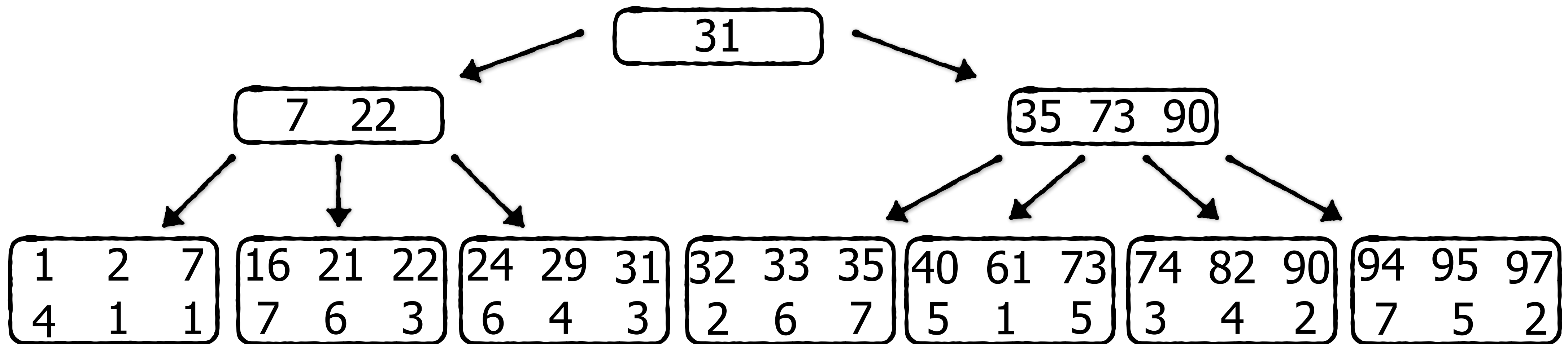
The next layer is of internal nodes, containing a delimiting key to separate between each pair of adjacent leaves.

Apply this idea recursively.



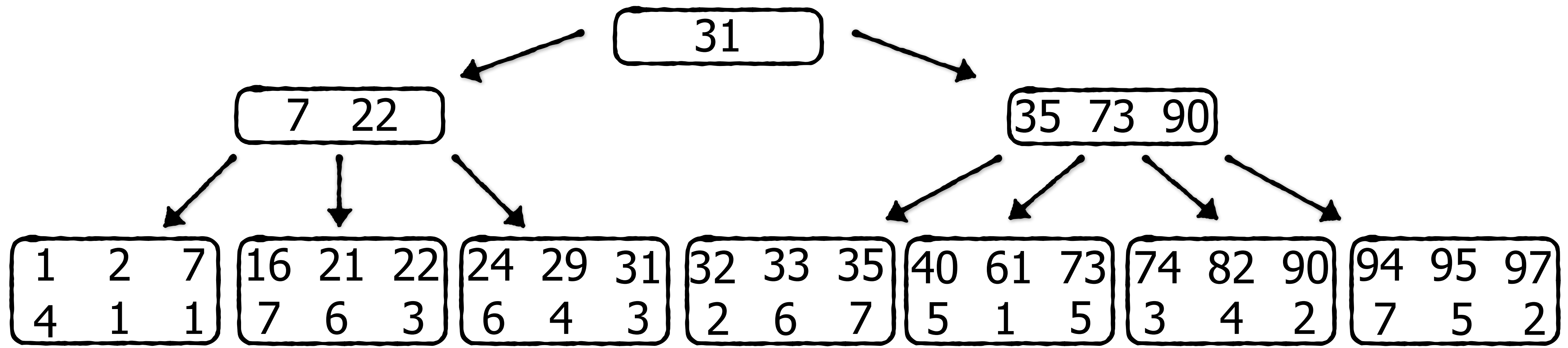
7			6			5			4			3			2			1		
16	35	94	24	33	21	73	40	95	1	82	29	22	74	31	32	90	97	2	7	61
...			...			...			...			...			...			...		

How to search?

[illegible]

How to search? Binary search each node from root and follow matching pointer

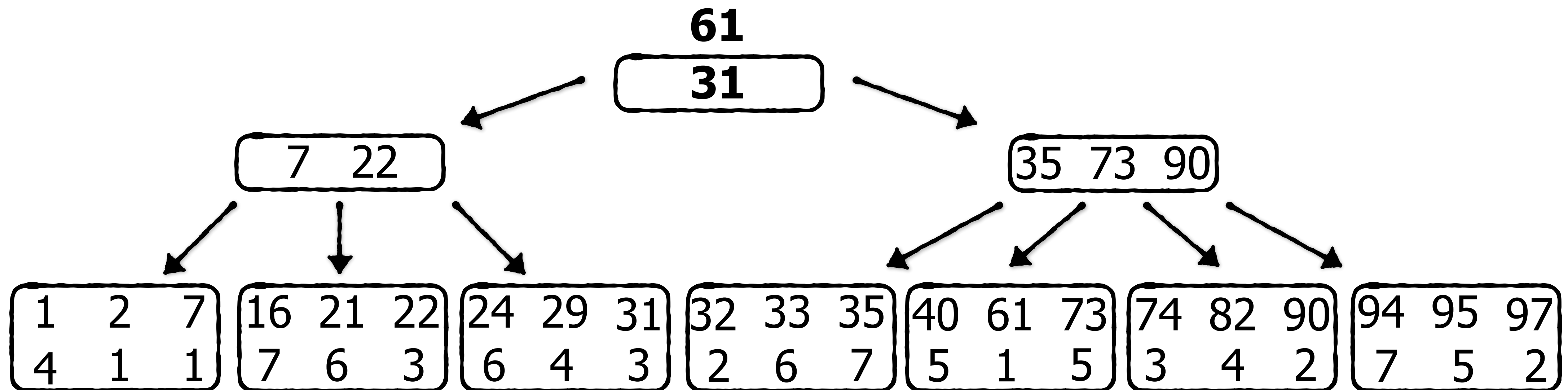
For example, search for **61**



7			6			5			4			3			2			1		
16	35	94	24	33	21	73	40	95	1	82	29	22	74	31	32	90	97	2	7	61
...			...			...			...			...			...			...		

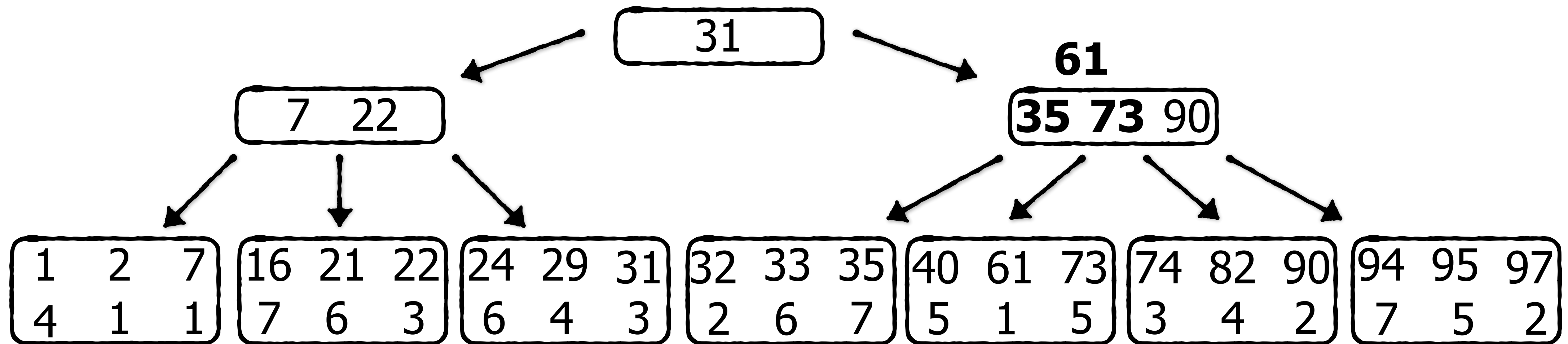
Page #
Table

How to search? Binary search each node from root and follow matching pointer

[illegible]

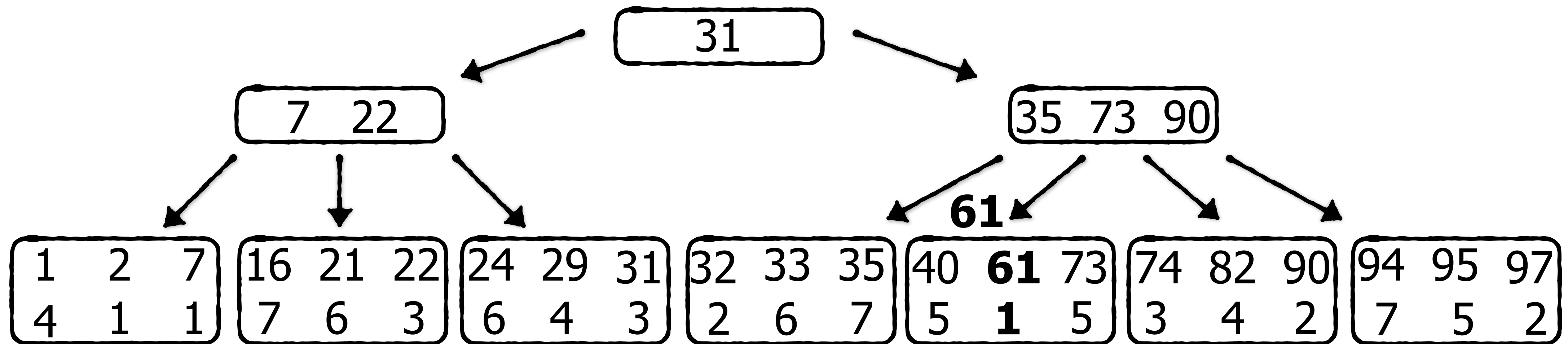
Page #

How to search? Binary search each node from root and follow matching pointer

[illegible]

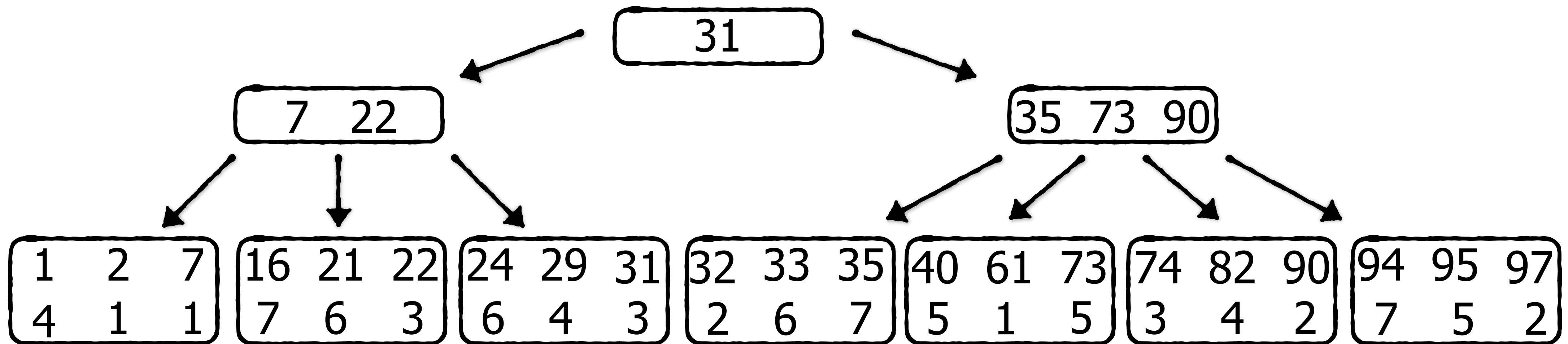
Page #

How to search? Binary search each node from root and follow matching pointer



7	6	5	4	3	2	1	Page #
16 35 94	24 33 21	73 40 95	1 82 29	22 74 31	32 90 97	2 7 61	
...	...	...	...	...	...	...	

How to search? Binary search each node from root and follow matching pointer



7

6

5

4

3

2

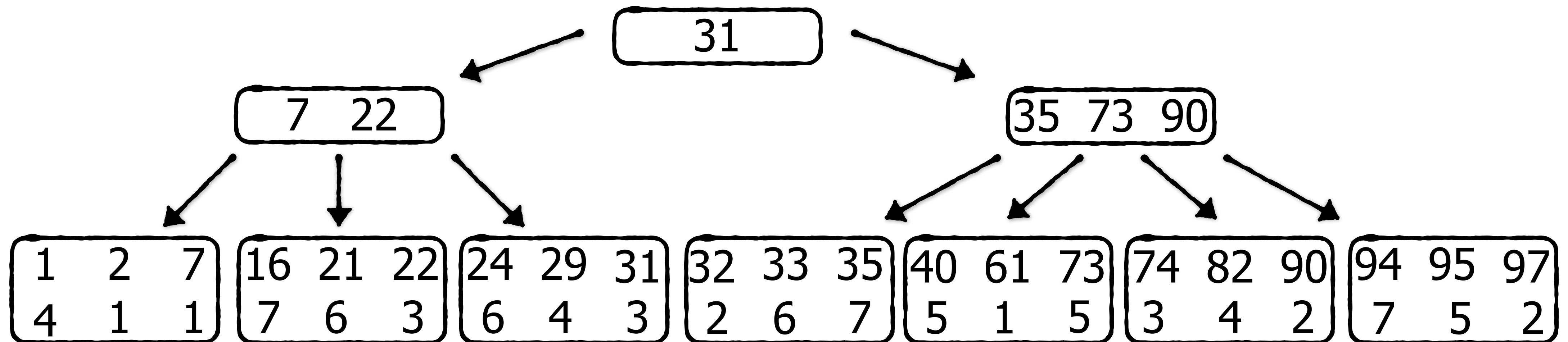
1

61

Page #

[illegible]

Search cost:



7

6

5

4

3

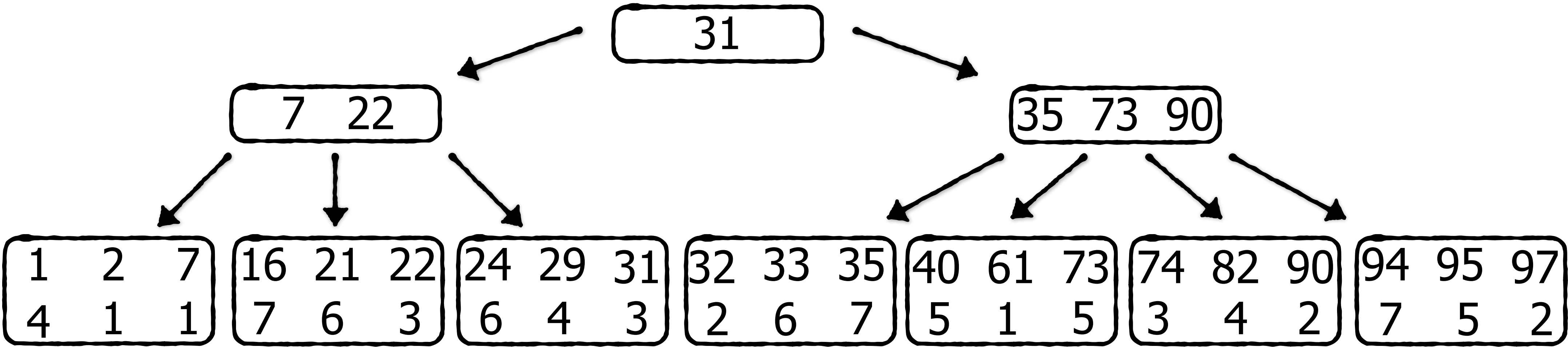
2

1

Page #

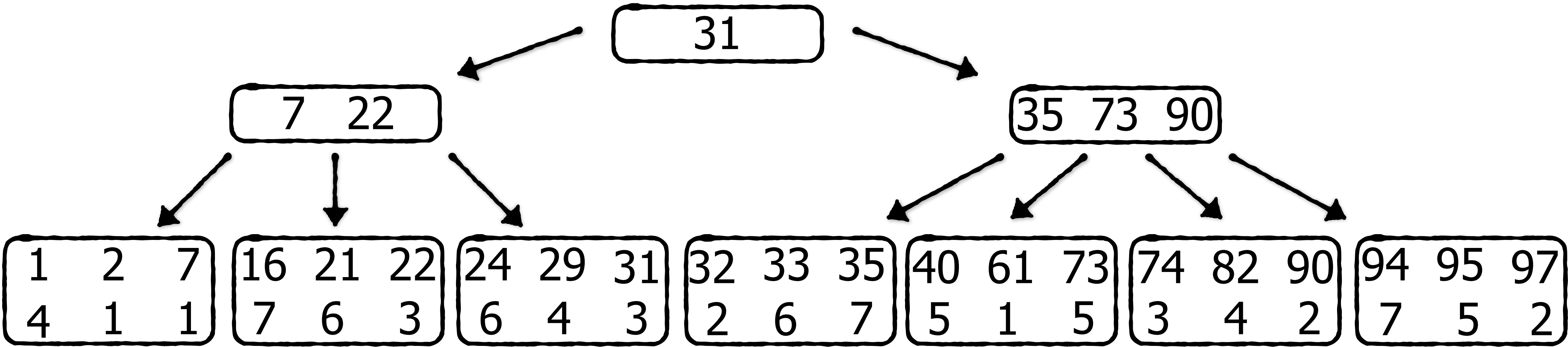
[illegible]

Search cost: = **depth of tree**
 = $\log_B(N) - 1$
 ↗ ↖
 # fanout **# leaves**



7	6	5	4	3	2	1	Page #
16 35 94	24 33 21	73 40 95	1 82 29	22 74 31	32 90 97	2 7 61	
...	...	...	...	...	...	...	

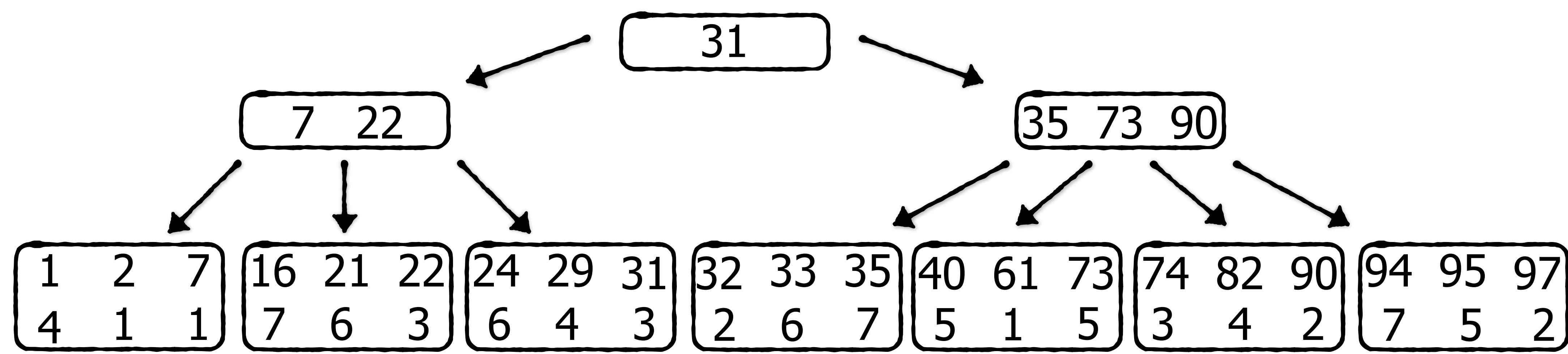
Search cost: **= depth of tree**
 = $O(\log_B N)$ I/O



7			6			5			4			3			2			1			Page #
16	35	94	24	33	21	73	40	95	1	82	29	22	74	31	32	90	97	2	7	61	
...			...			...			...			...			...			...			

Search cost: $O(\log_B N)$ I/O

B is ≈ 100 -1000, so a B-tree is shallow in practice

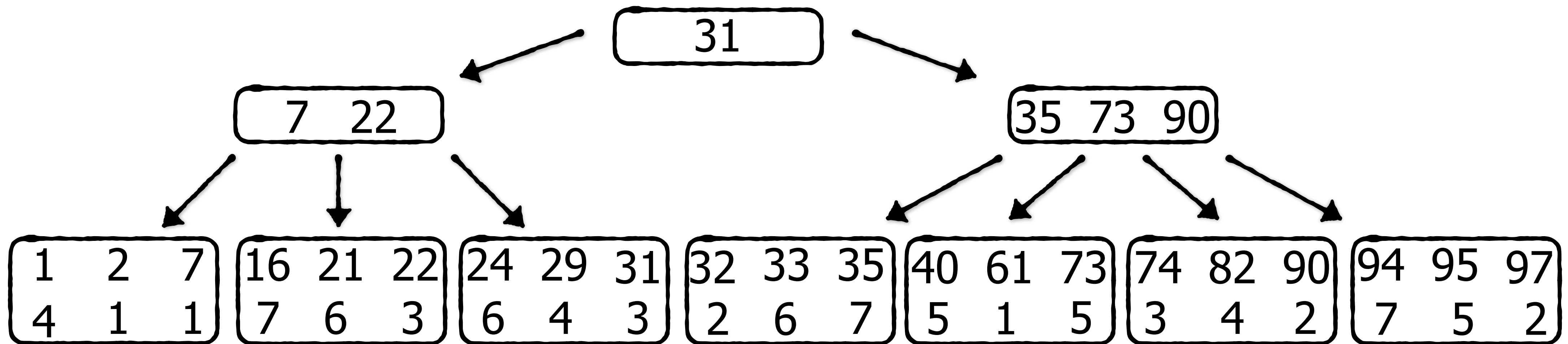


Page #						
7	6	5	4	3	2	1
16 35 94	24 33 21	73 40 95	1 82 29	22 74 31	32 90 97	2 7 61
...	...	...	...	...	...	...

Search cost: $O(\log_B N)$ I/O

B is ≈ 100 -1000, so a B-tree is shallow in practice

For example, if $N=2^{30}$ and $B=2^{10}$, what is the depth?

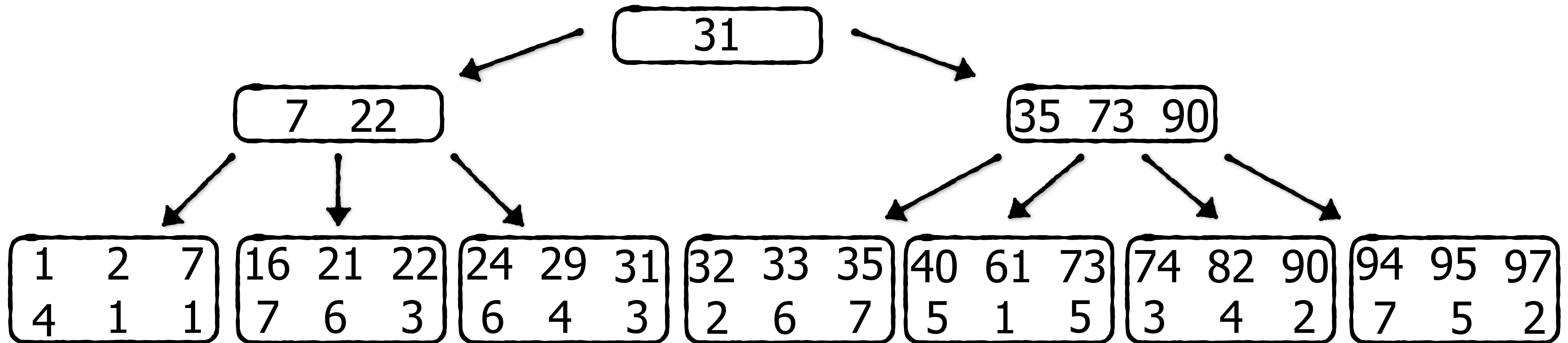


7	6	5	4	3	2	1	Page #
16 35 94	24 33 21	73 40 95	1 82 29	22 74 31	32 90 97	2 7 61	
...	...	...	...	...	...	...	

Search cost: $O(\log_B N)$ I/O

B is $\approx 100-1000$, so a B-tree is shallow in practice

For example, if $N=2^{30}$ and $B=2^{10}$, what is the depth? **$\log_B N \approx 3$**



7

6

5

4

3

2

1

Page #

16 35 94:24 33 21:73 40 95:1 82 29:22 74 31:32 90 97:2 7 61

■ ■ ■

■ ■ ■

■ ■ ■

■ ■ ■

■ ■ ■

□ □ □

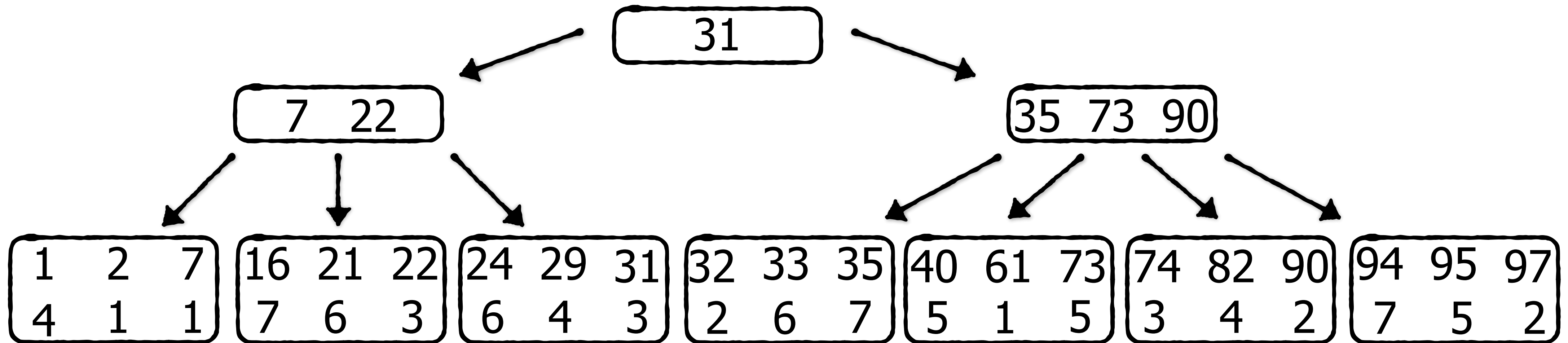
■ ■ ■

Search cost: $O(\log_B N)$ I/O

B is $\approx 100-1000$, so a B-tree is shallow in practice

For example, if $N=2^{30}$ and $B=2^{10}$, what is the depth? $\log_B N \approx 3$

With a binary tree, it would have been far larger.



7

6

5

4

3

2

1

Page #

16 35 94:24 33 21:73 40 95:1 82 29:22 74 31:32 90 97:2 7 61

■ ■ ■

■ ■ ■

■ ■ ■

■ ■ ■

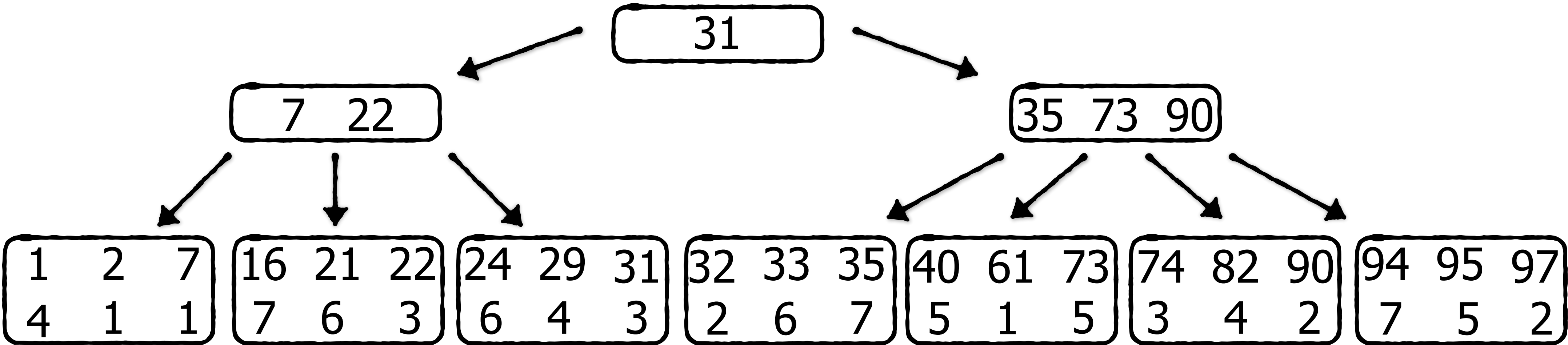
■ ■ ■

□ □ □

■ ■ ■

Search cost: $O(\log_B N)$ I/O

How many CPU comparisons?



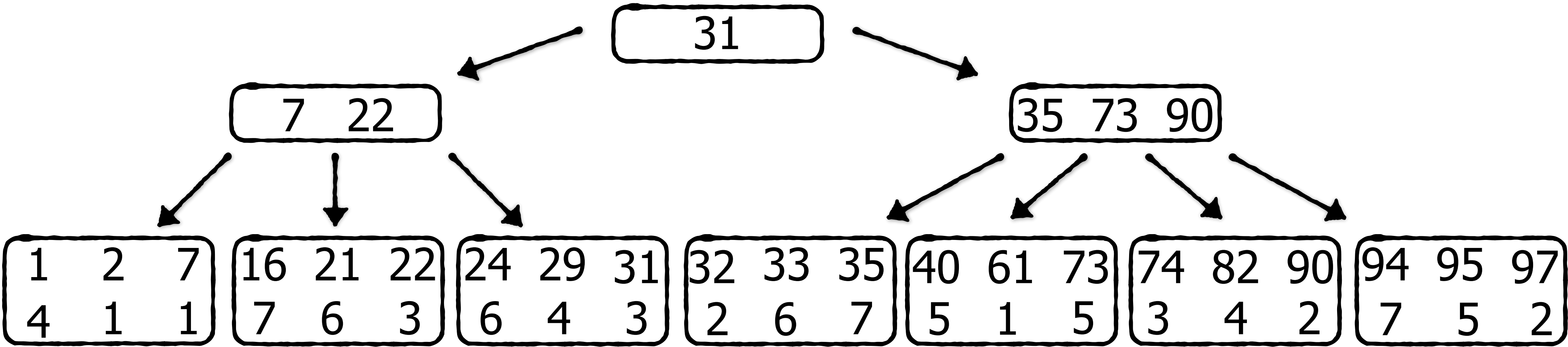
7	6	5	4	3	2	1	Page #
16 35 94	24 33 21	73 40 95	1 82 29	22 74 31	32 90 97	2 7 61	
...	...	...	...	...	...	...	

Search cost: $O(\log_B N)$ I/O

How many CPU comparisons?

$O(\log_B N * \log_2 B) = O(\log_2 N)$

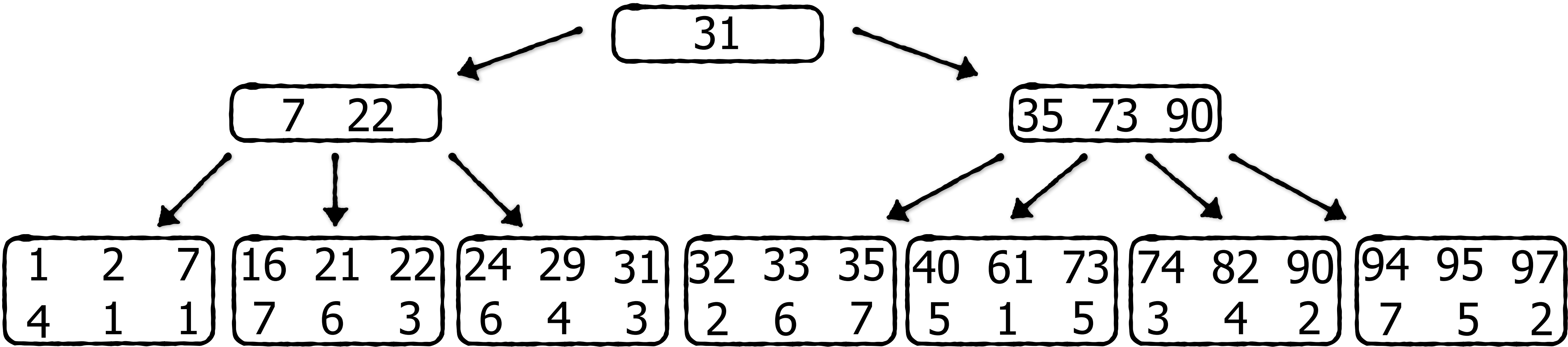
$\uparrow$ $\uparrow$
pages **binary**
searched **search**



7			6			5			4			3			2			1			Page #
16	35	94	24	33	21	73	40	95	1	82	29	22	74	31	32	90	97	2	7	61	
...			...			...			...			...			...			...			

Search cost: $O(\log_B N)$ I/O

How many CPU comparisons? **$O(\log_2 N)$**



Page #						
7	6	5	4	3	2	1
16 35 94	24 33 21	73 40 95	1 82 29	22 74 31	32 90 97	2 7 61
...	...	...	...	...	...	...

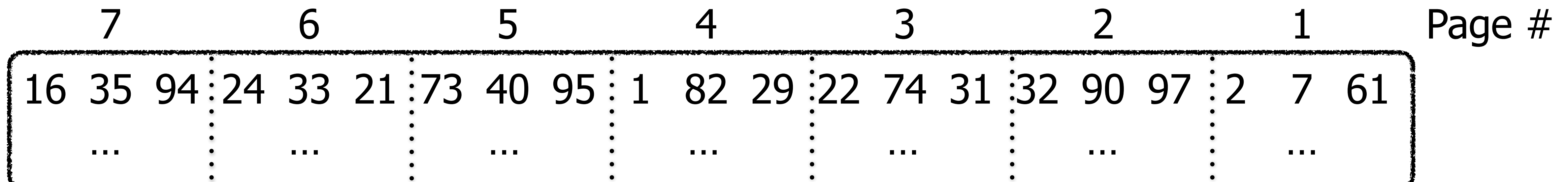
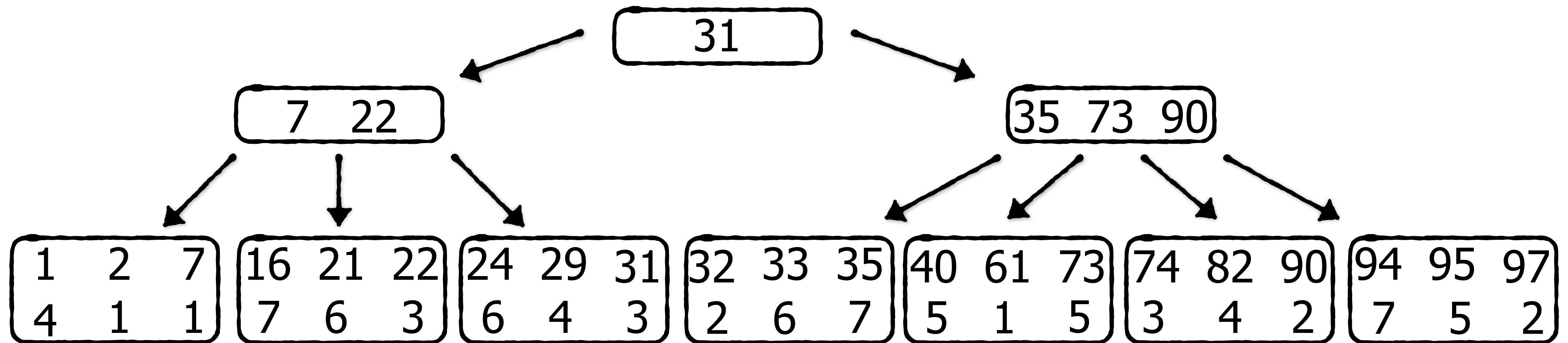
Both are a part of search cost

Search cost: $O(\log_B N)$ I/O

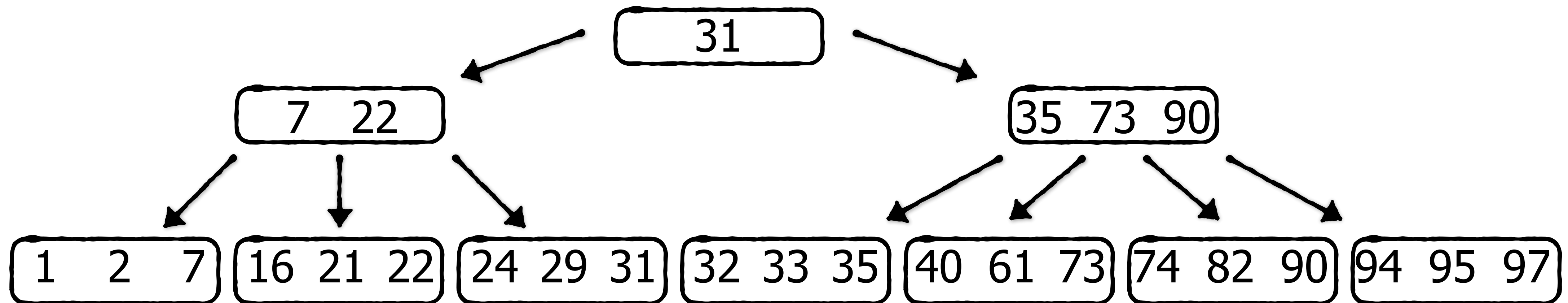
← But this is the bottleneck

How many CPU comparisons? $O(\log_2 N)$

← And this usually isn't



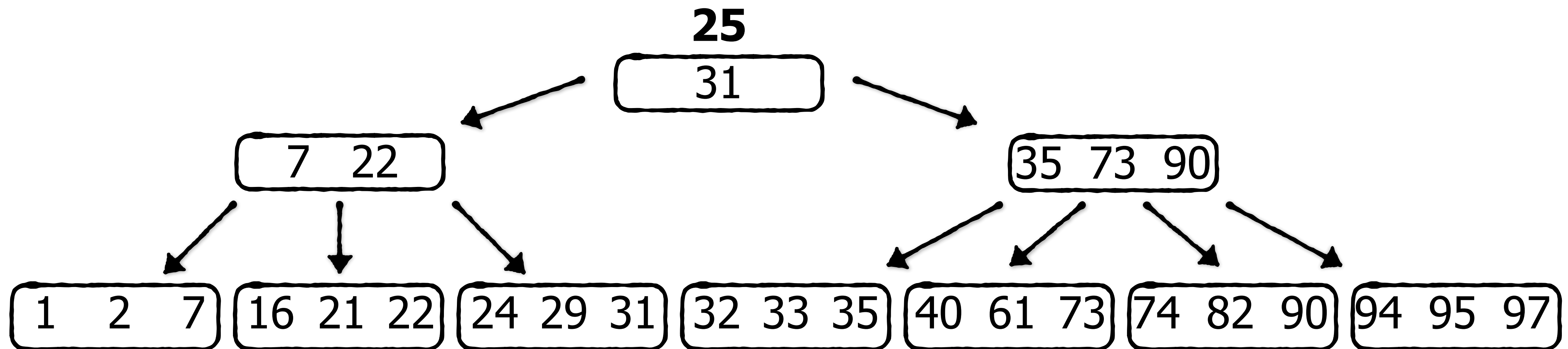
How to insert? e.g., key 25



Proposal 4: B-Tree

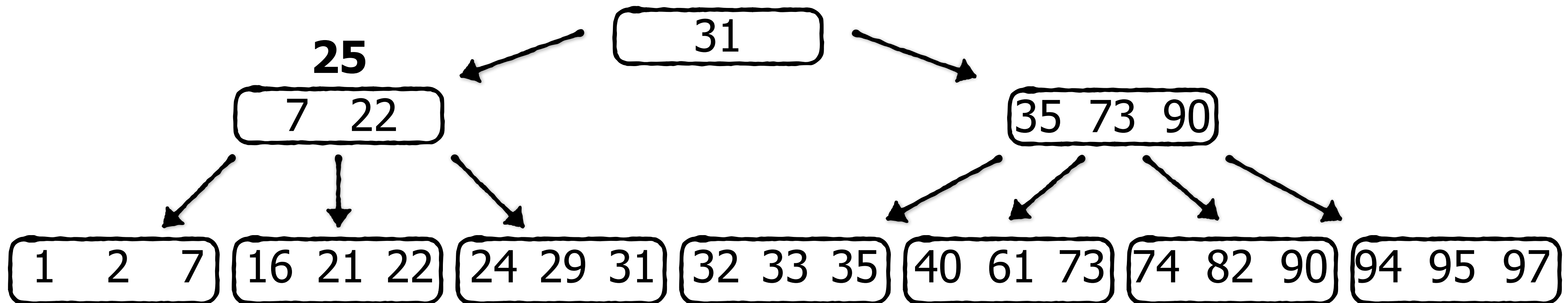
How to insert?

First find target node



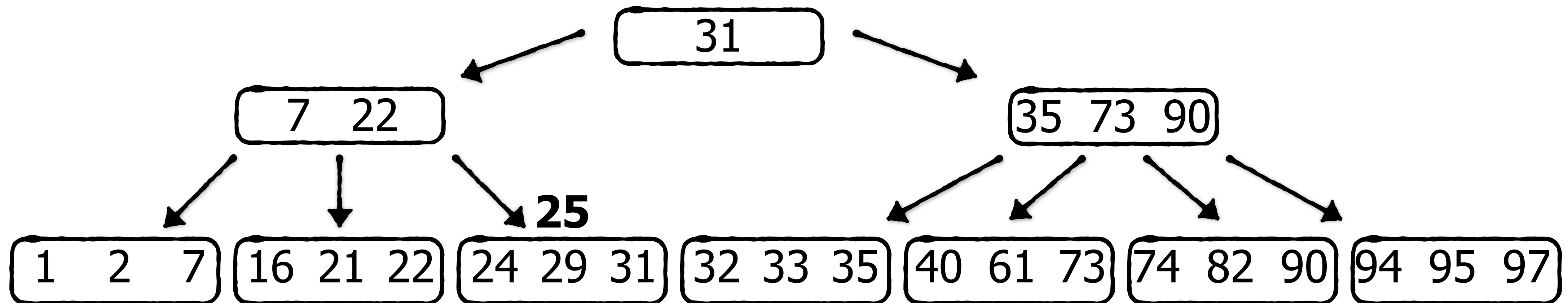
How to insert?

First find target node



How to insert?

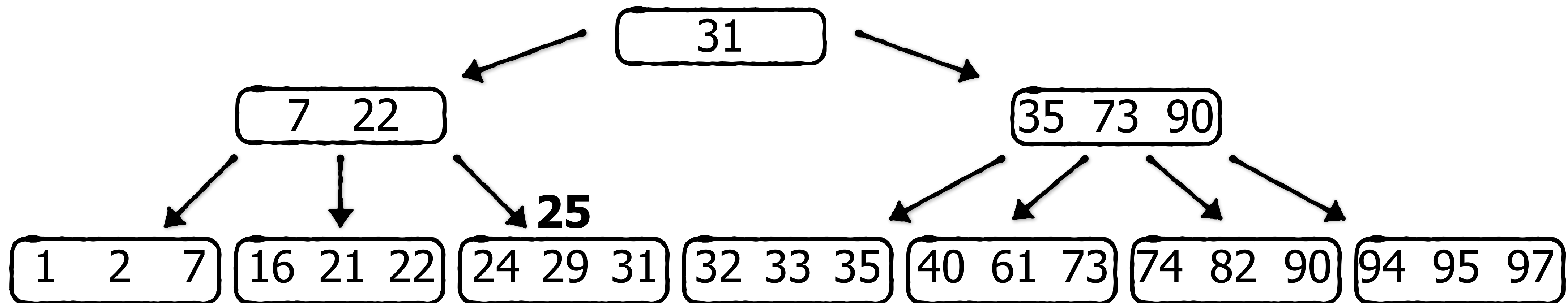
First find target node



How to insert?

First find target node

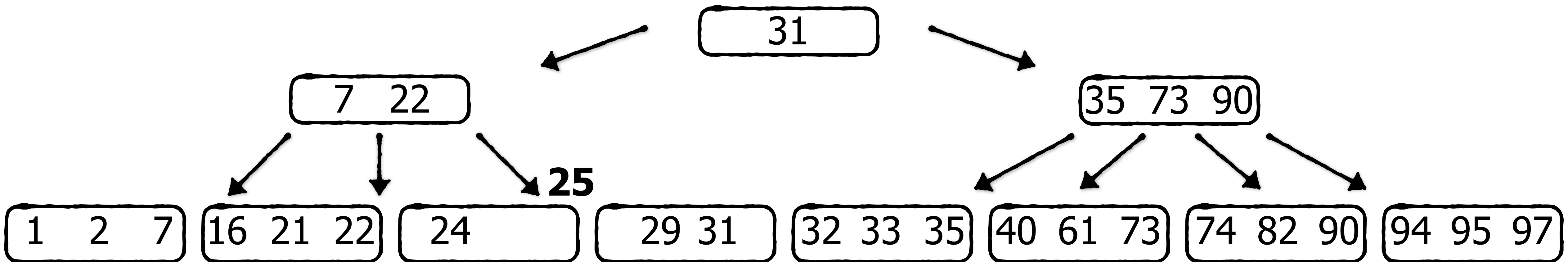
If node is full, split target node



How to insert?

First find target node

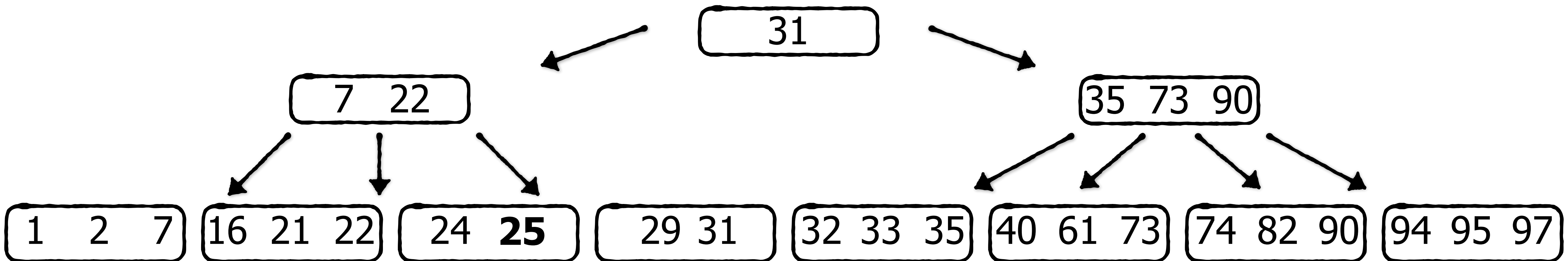
If node is full, split target node



How to insert?

First find target node

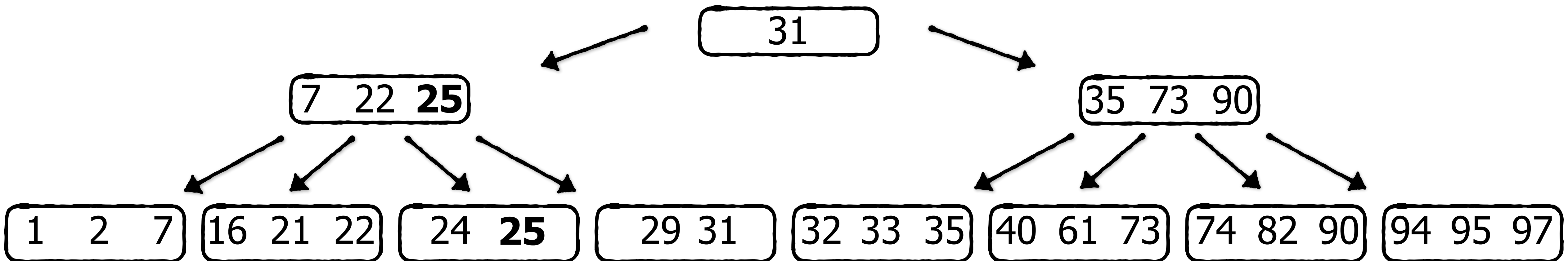
If node is full, split target node



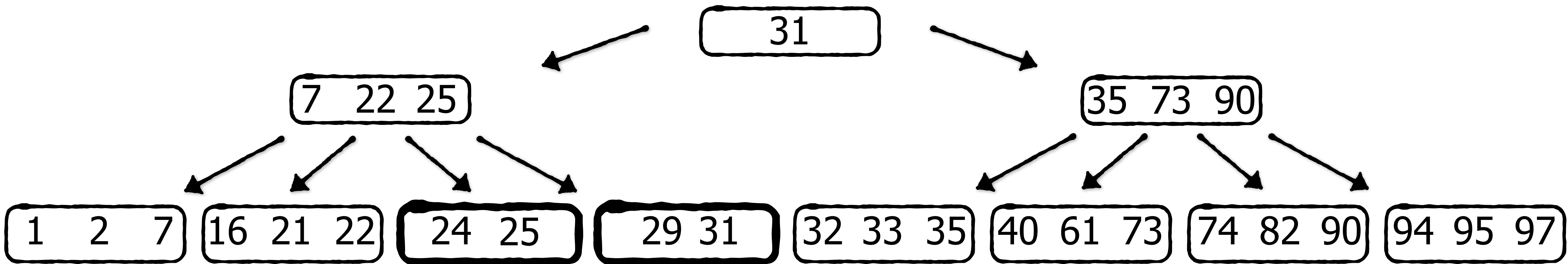
How to insert?

First find target node

Connect new node to parent



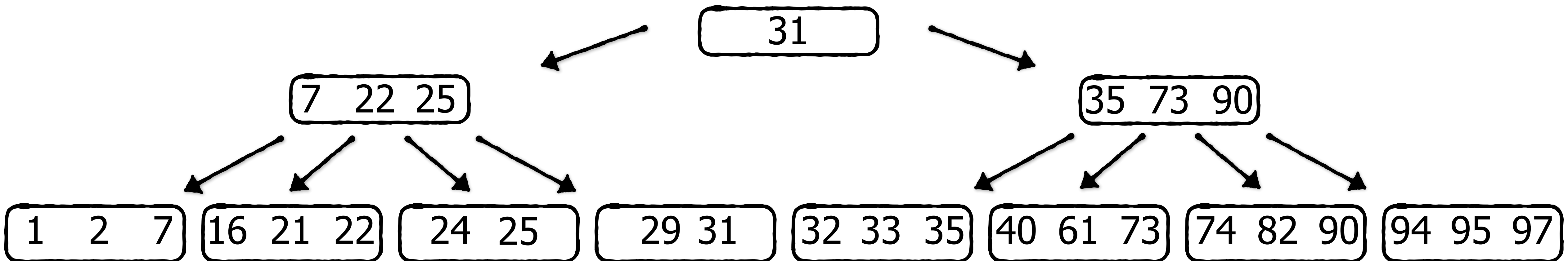
After a node splits, the two new nodes have $\approx 50\%$ free capacity (padding)



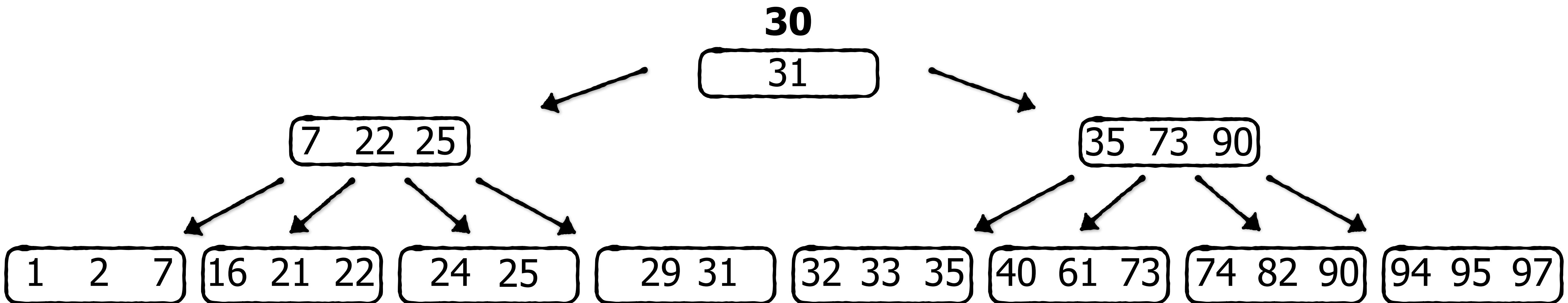
After a node splits, the two new nodes have $\approx 50\%$ free capacity (padding)

Hence, most inserts do not cause splits as leaf nodes usually have free space

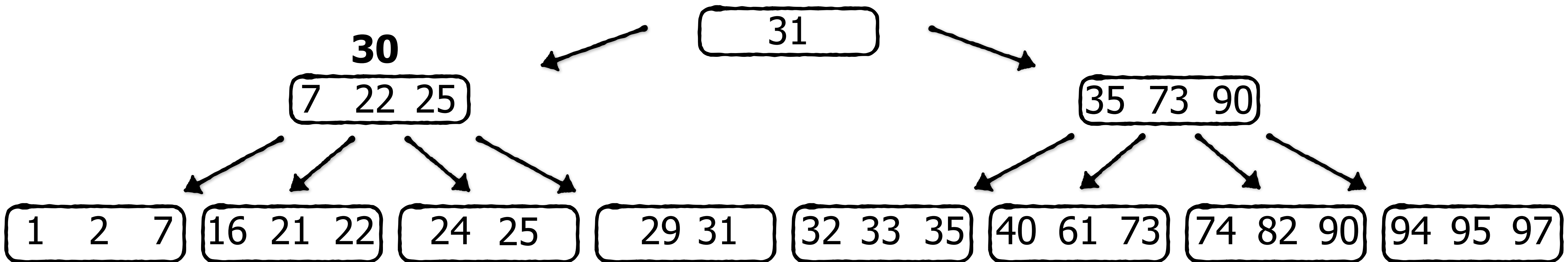
e.g., insert **30**



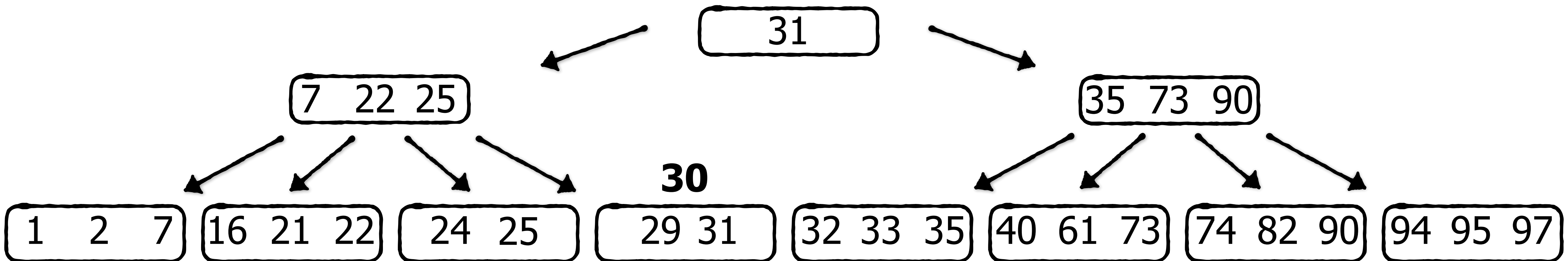
most inserts do not cause splits as leaf nodes usually have free space



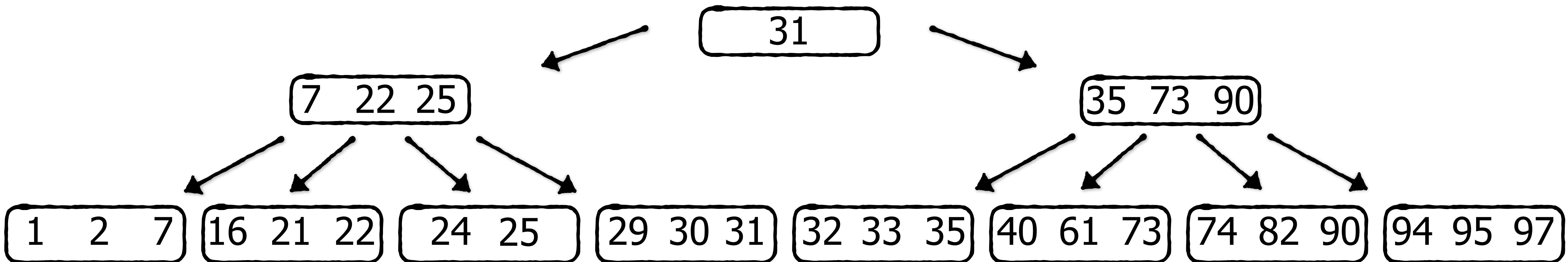
most inserts do not cause splits as leaf nodes usually have free space



most inserts do not cause splits as leaf nodes usually have free space



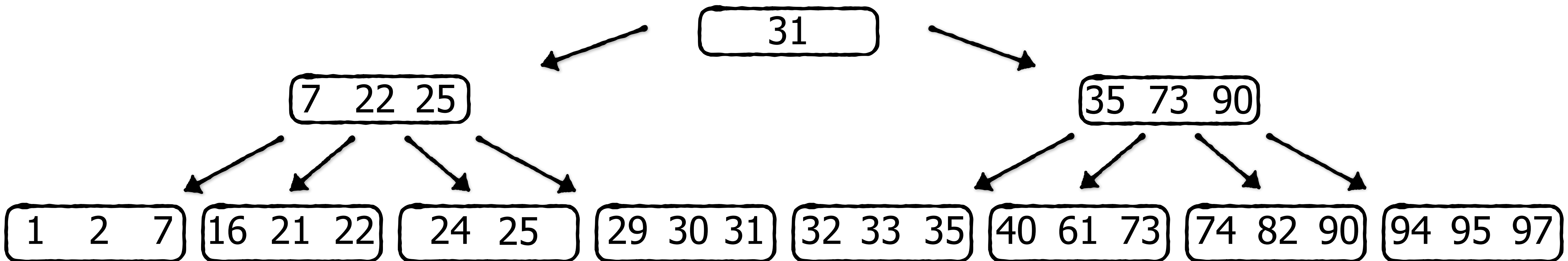
most inserts do not cause splits as leaf nodes usually have free space



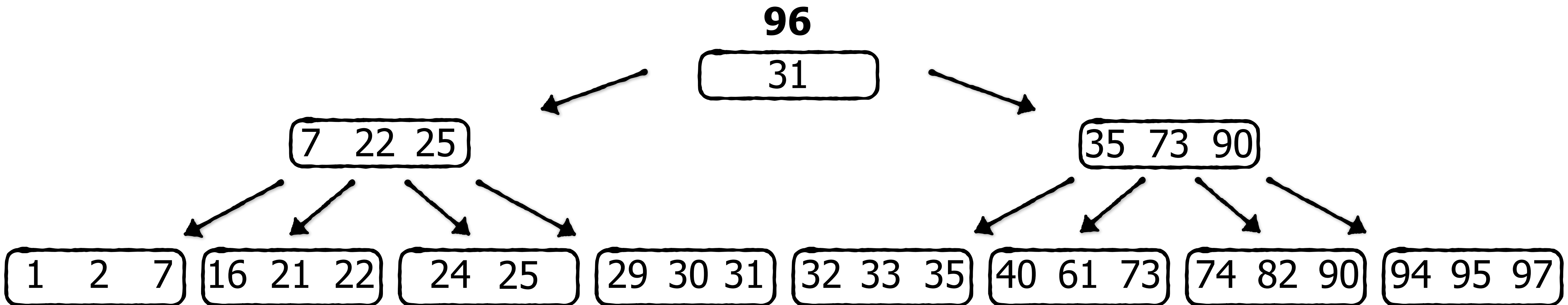
Hence, a B-tree takes up at most twice as much storage space as the entries it contains due to padding. In practice, 69-75% of data size.

Rarely, a split propagates upwards

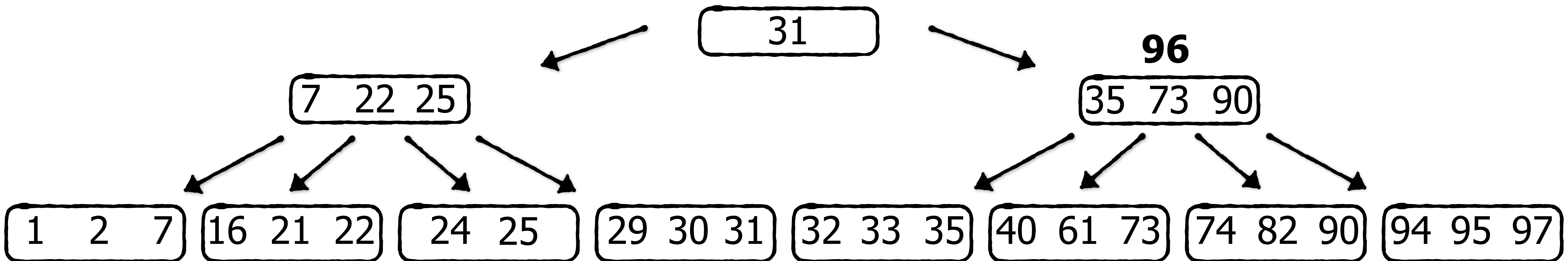
e.g., insert **96**



Rarely, a split propagates upwards

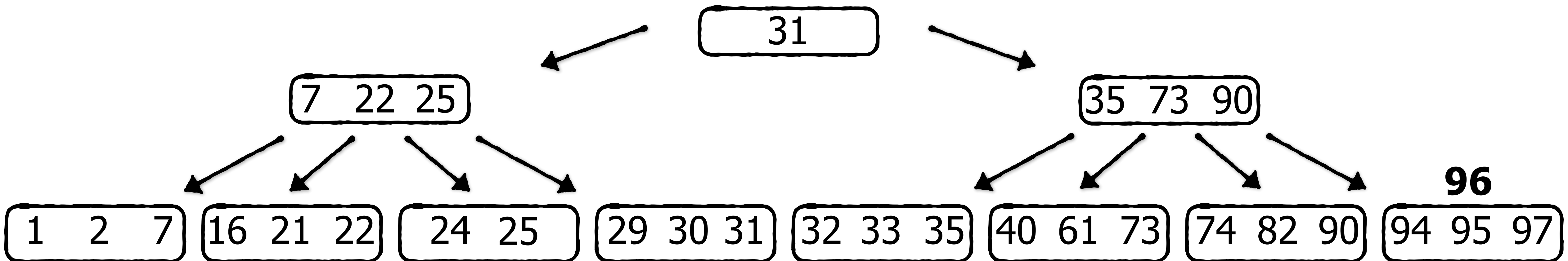


Rarely, a split propagates upwards



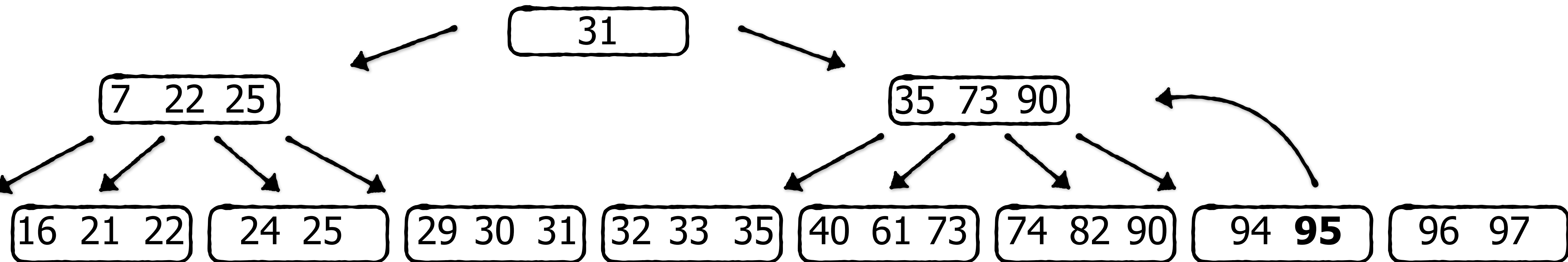
Rarely, a split propagates upwards

Split leaf



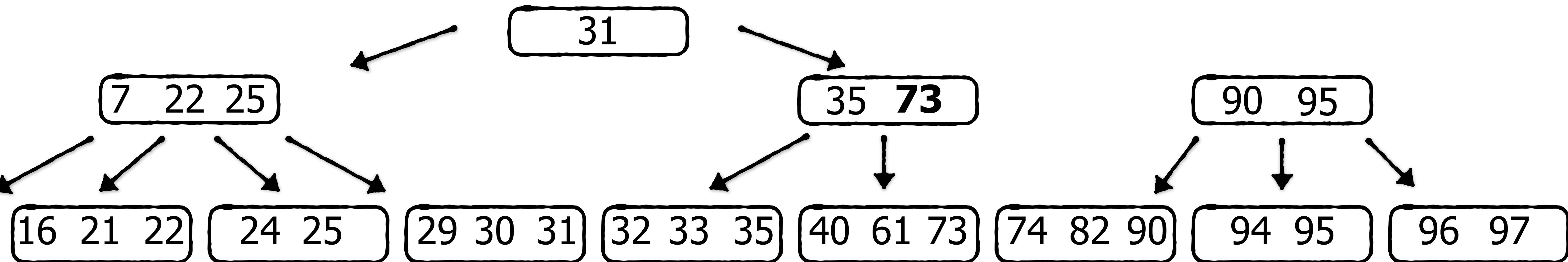
Rarely, a split propagates upwards

Create new parent

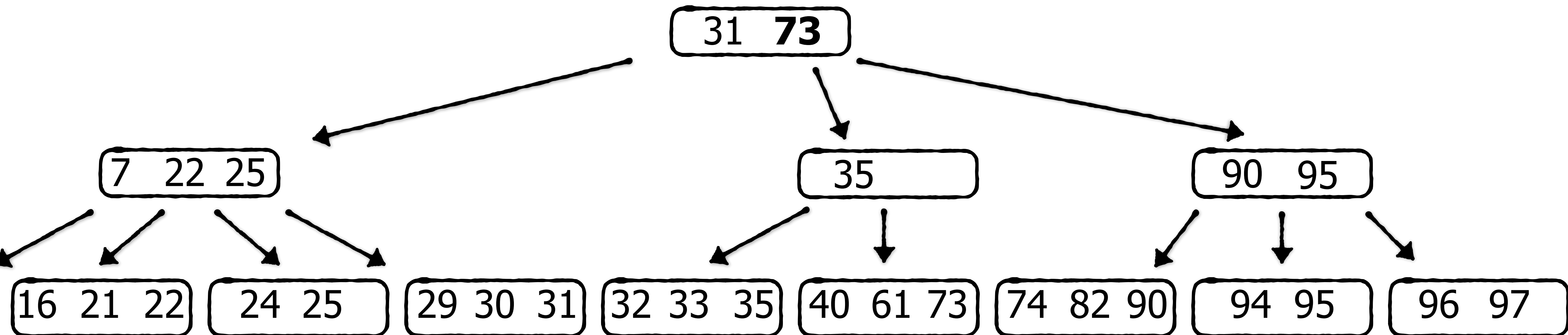


Rarely, a split propagates upwards

Connect new parent to root



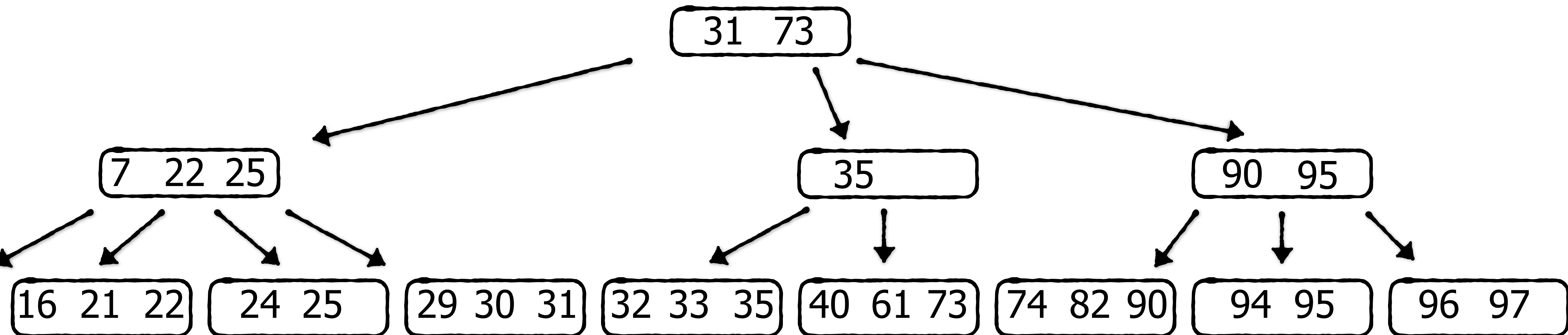
read I/Os per insertion?



read I/Os per insertion?

$O(\log_B N)$

write I/Os per insertion?



read I/Os per insertion?

$O(\log_B N)$

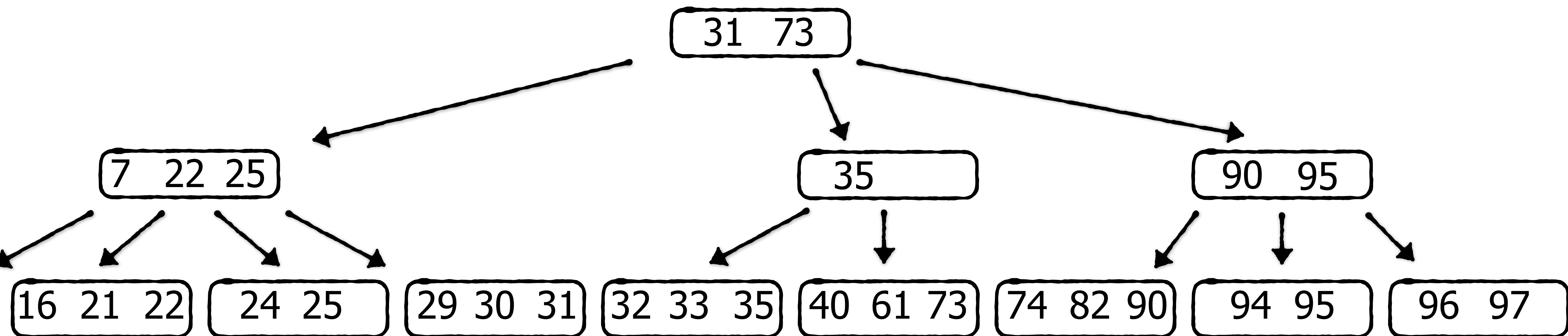
write I/Os per insertion?

Intuition: Every insertion updates 1 leaf

one in $O(B^{-1})$ insertions triggers leaf split

one in $O(B^{-2})$ insertions triggers parent split

one in $O(B^{-3})$ insertions triggers grandparent split

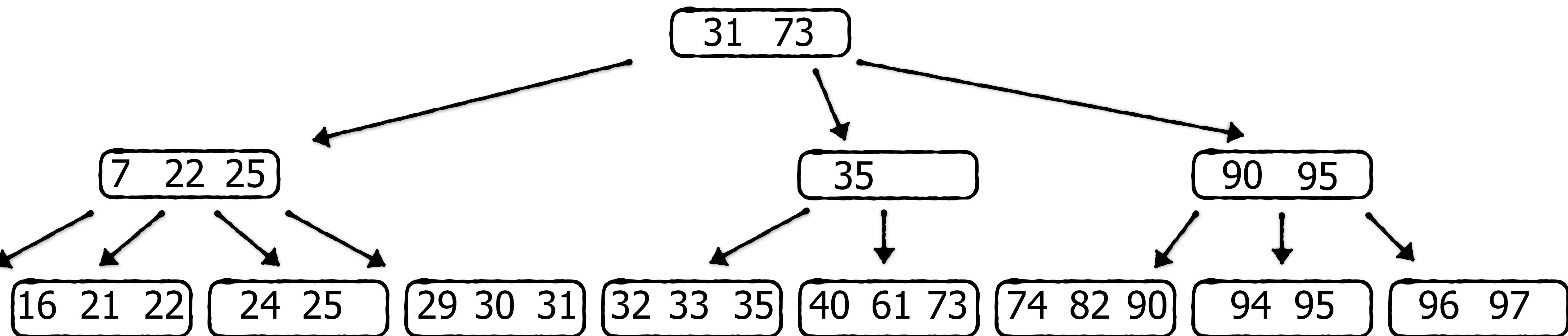


How many read I/Os per insertion?

$O(\log_B N)$

How many write I/Os per insertion?

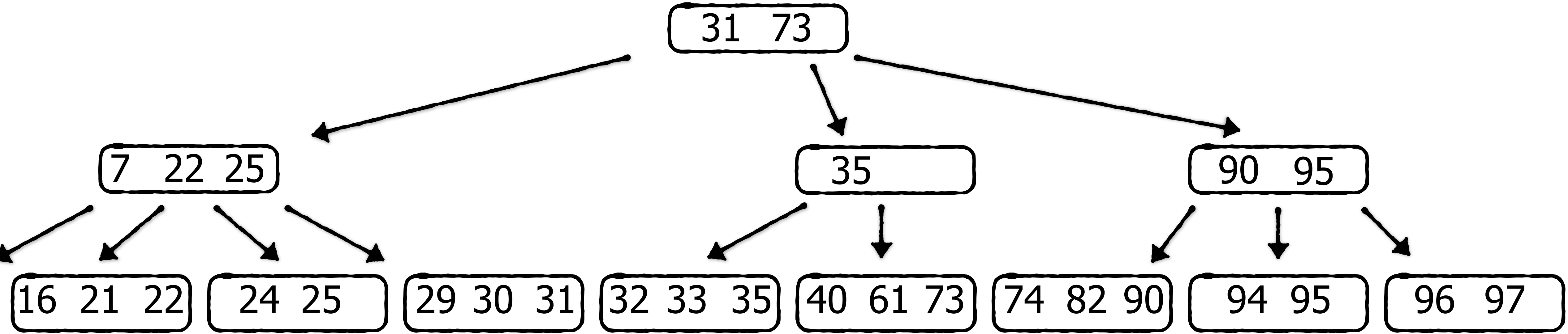
$$1 + B^{-1} + B^{-2} + B^{-3} + \dots = O(1)$$



An insertion costs

$O(\log_B N)$
 $O(1)$

Read I/Os
Write I/O



An insertion costs

$O(\log_B N)$

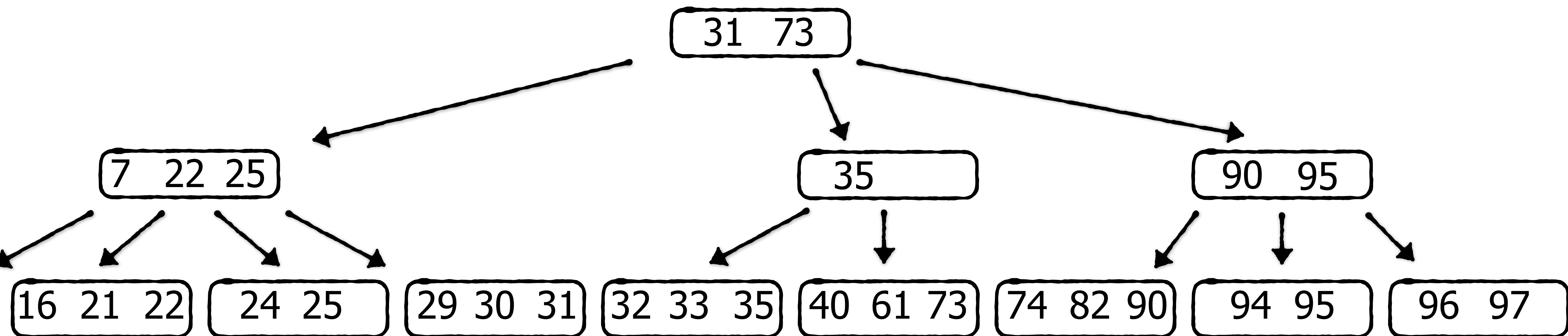
Read I/Os

$O(1)$

Write I/O

On disk, read/write costs are symmetric.

On SSD, they are different, so it's important to differentiate.



An insertion costs

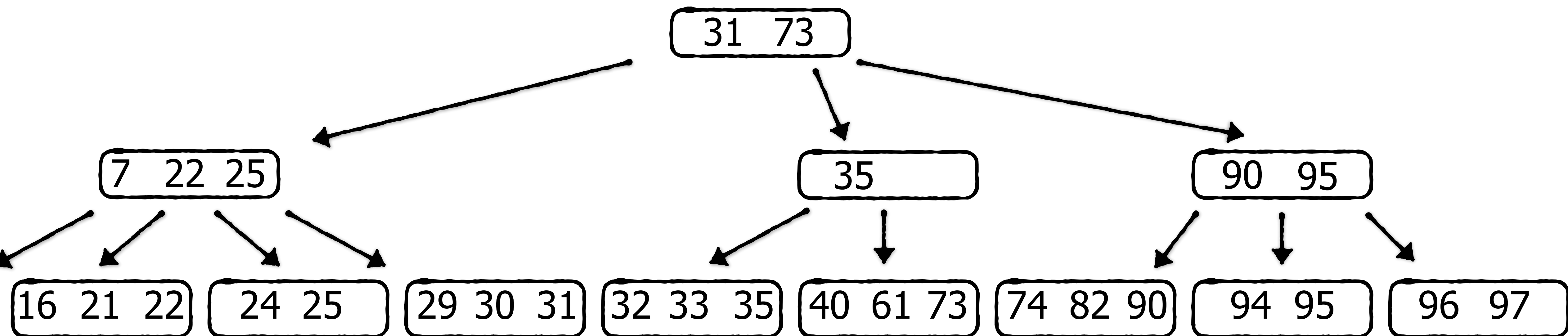
$O(\log_B N)$

Read I/Os

$O(1)$

Write I/O

Supposing internal nodes reside in the buffer pool, how does this model change?



An insertion costs

$O(1)$

Read I/Os

$O(1)$

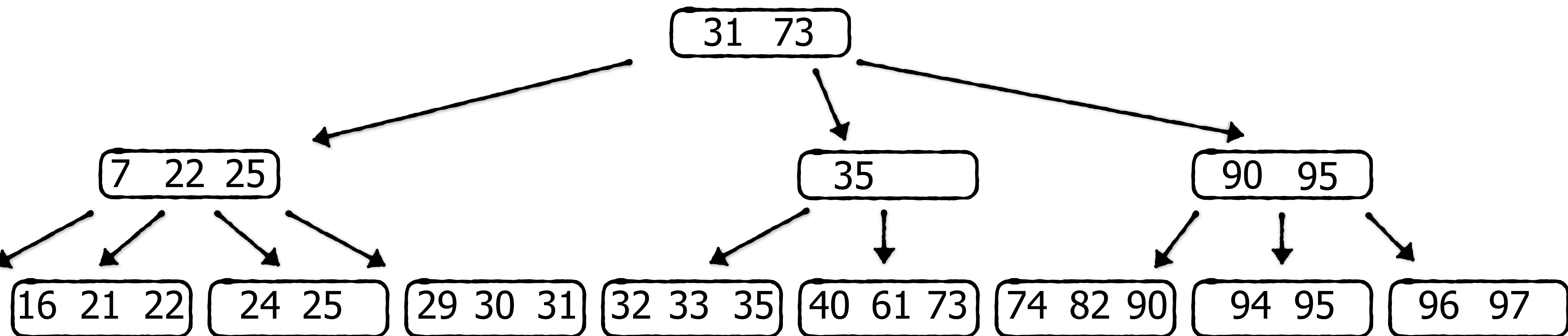
Write I/O

$O(N/B)$

Memory (# keys stored in memory)

Only one read I/O is needed to read a leaf per lookup.

If the B-tree contains $O(N)$ keys, $O(N/B)$ are stored in memory.



An insertion costs

$O(1)$

Read I/Os

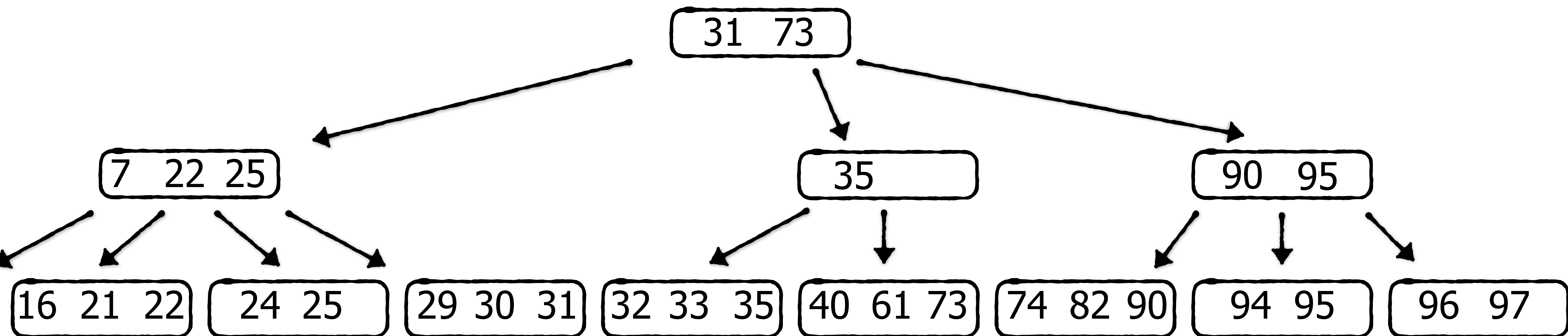
$O(1)$

Write I/O

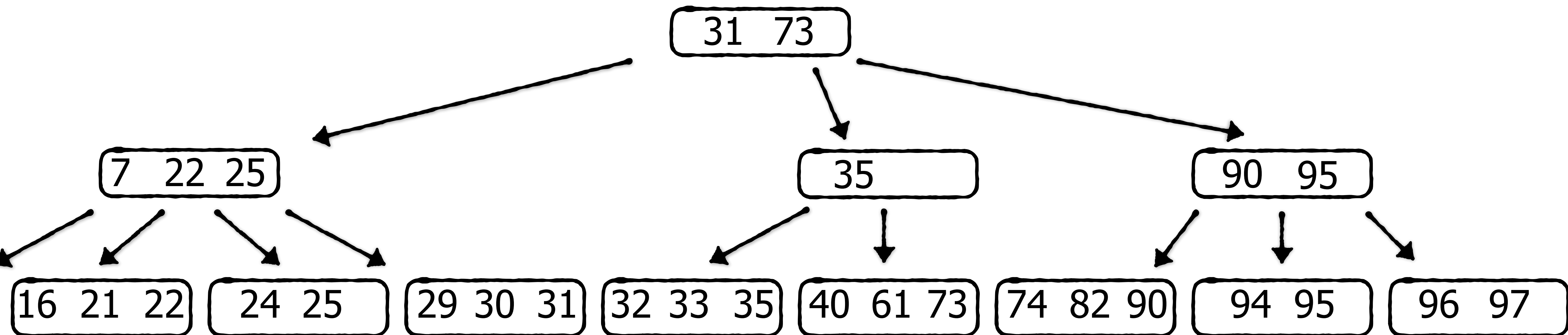
$O(N/B)$

Memory

In practice, internal nodes are usually in the buffer as they are accessed more and there are fewer of them.



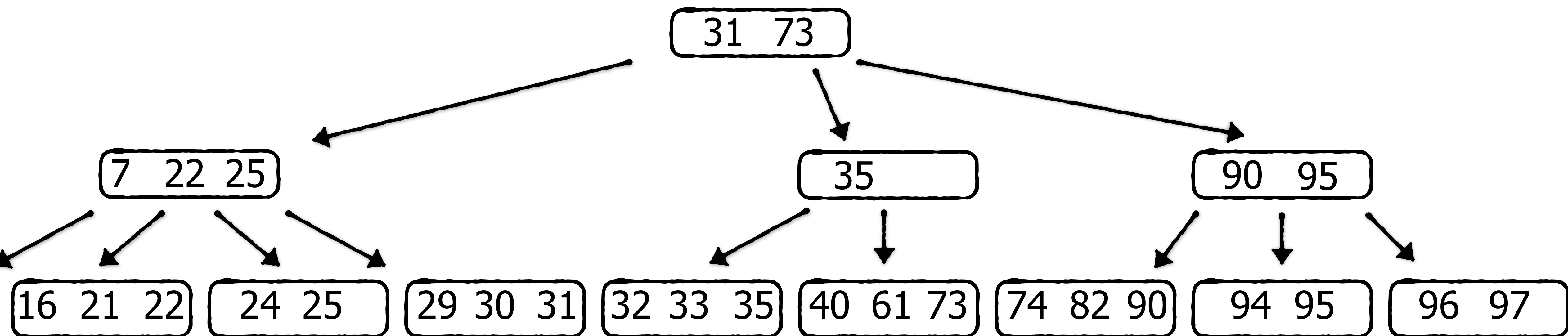
Indexes speed up queries. Do they have downsides?



Indexes speed up queries. Do they have downsides?

(1) They take up storage space

(2) They must be updated as data changes

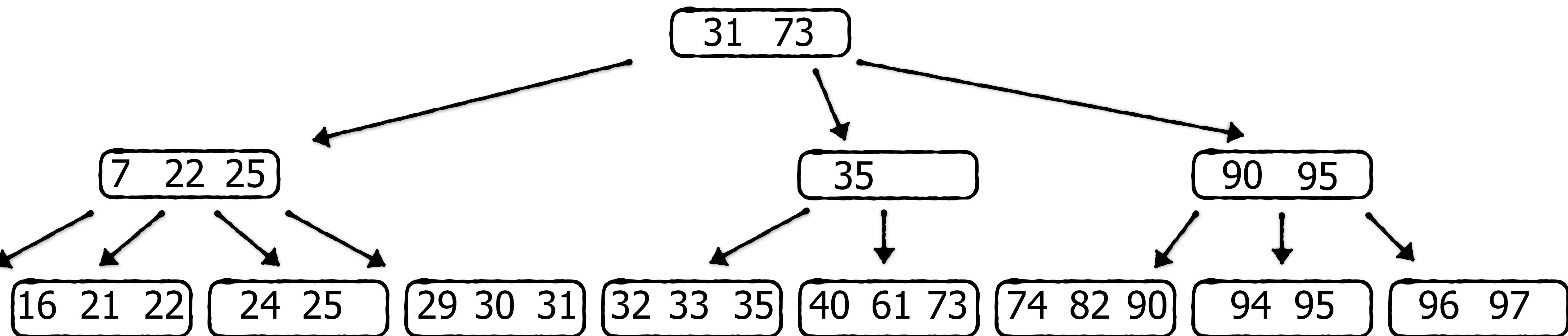


Indexes speed up queries. Do they have downsides?

- (1) They take up storage space
- (2) They must be updated as data changes

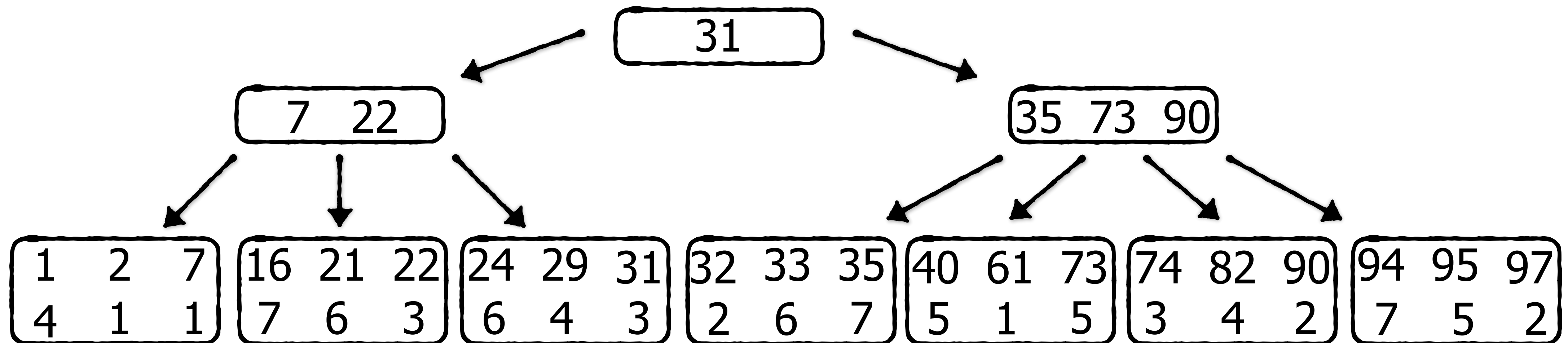
To index or not to index? That depends on the workload.

More on Thursday



How do range scans work?

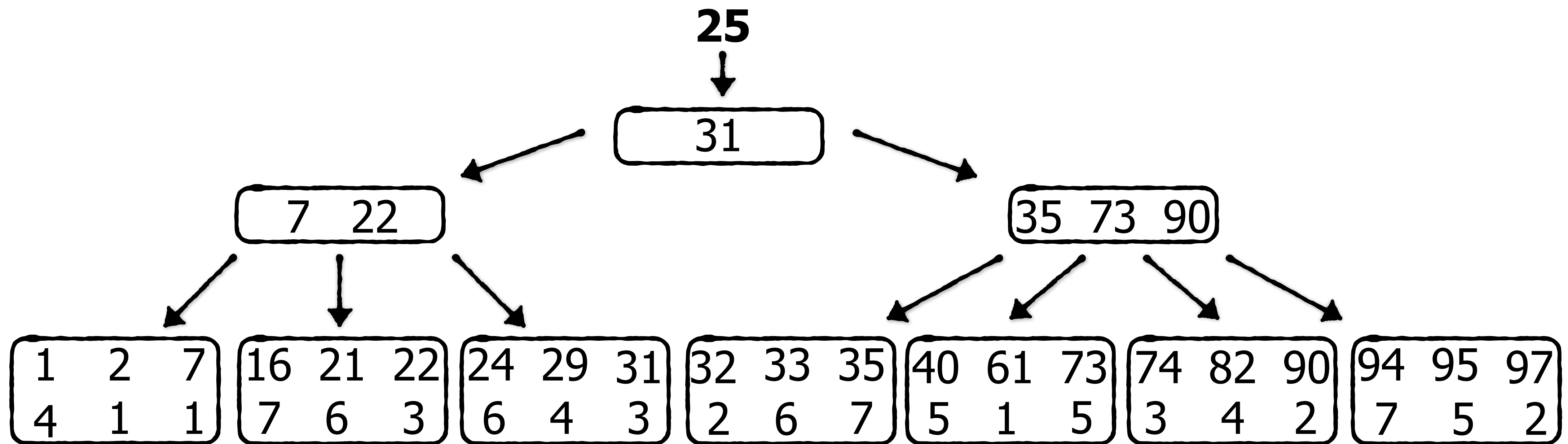
Select * from table where A > 25 and A < 35



7			6			5			4			3			2			1			Page #						
16	35	94	:	24	33	21	:	73	40	95	:	1	82	29	:	22	74	31	:	32	90	97	:	2	7	61	
...			:	...			:	...			:	...			:	...			:	...			:	...			

How do range scans work?

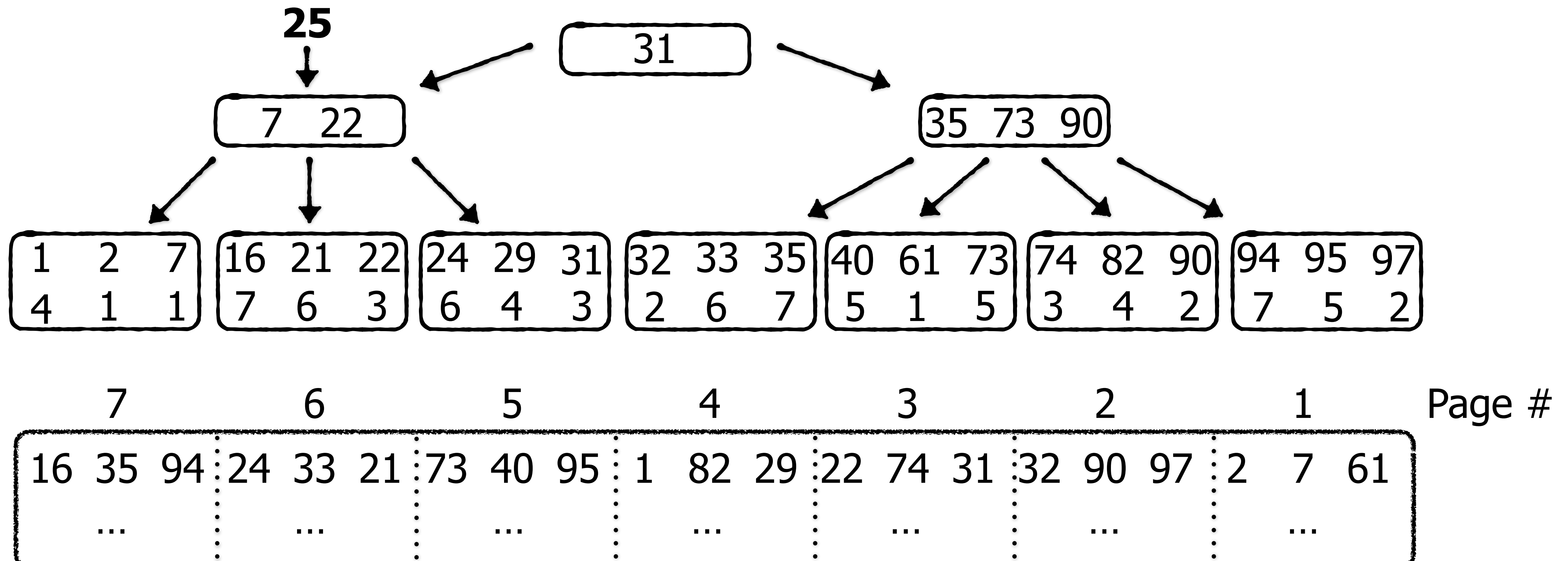
Select * from table where $A > 25$ and $A < 35$



7			6			5			4			3			2			1			Page #
16	35	94	24	33	21	73	40	95	1	82	29	22	74	31	32	90	97	2	7	61	
...			...			...			...			...			...			...			

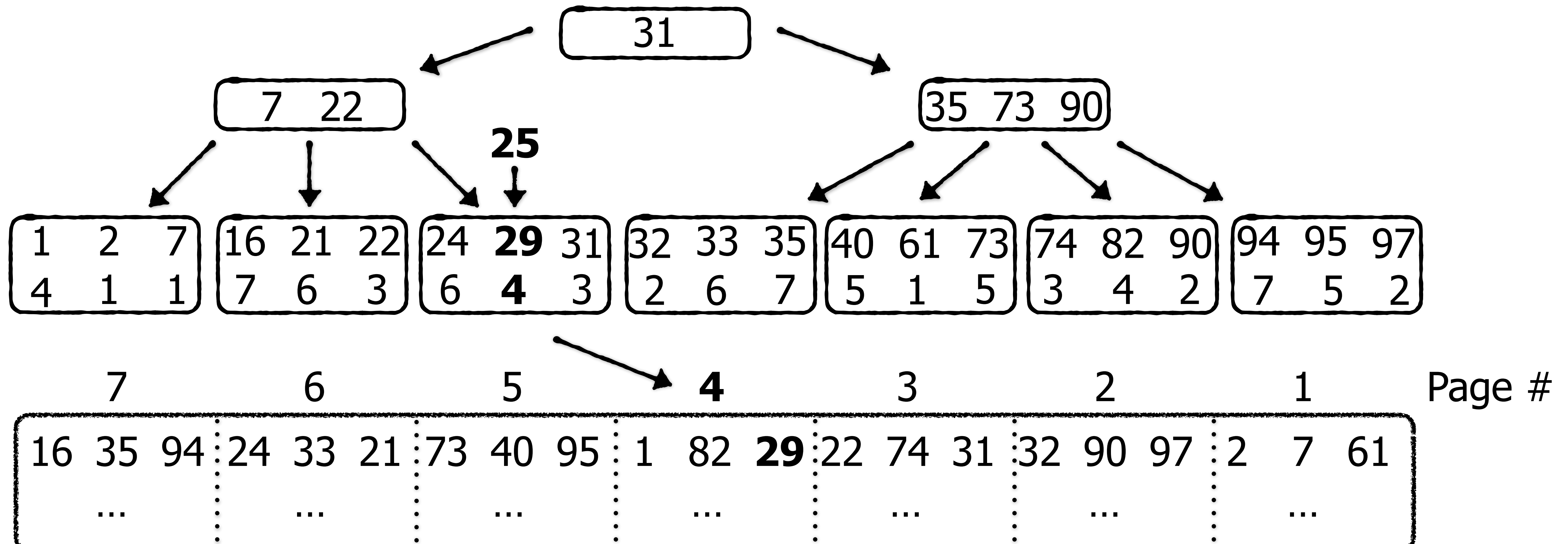
How do range scans work?

Select * from table where $A > 25$ and $A < 35$



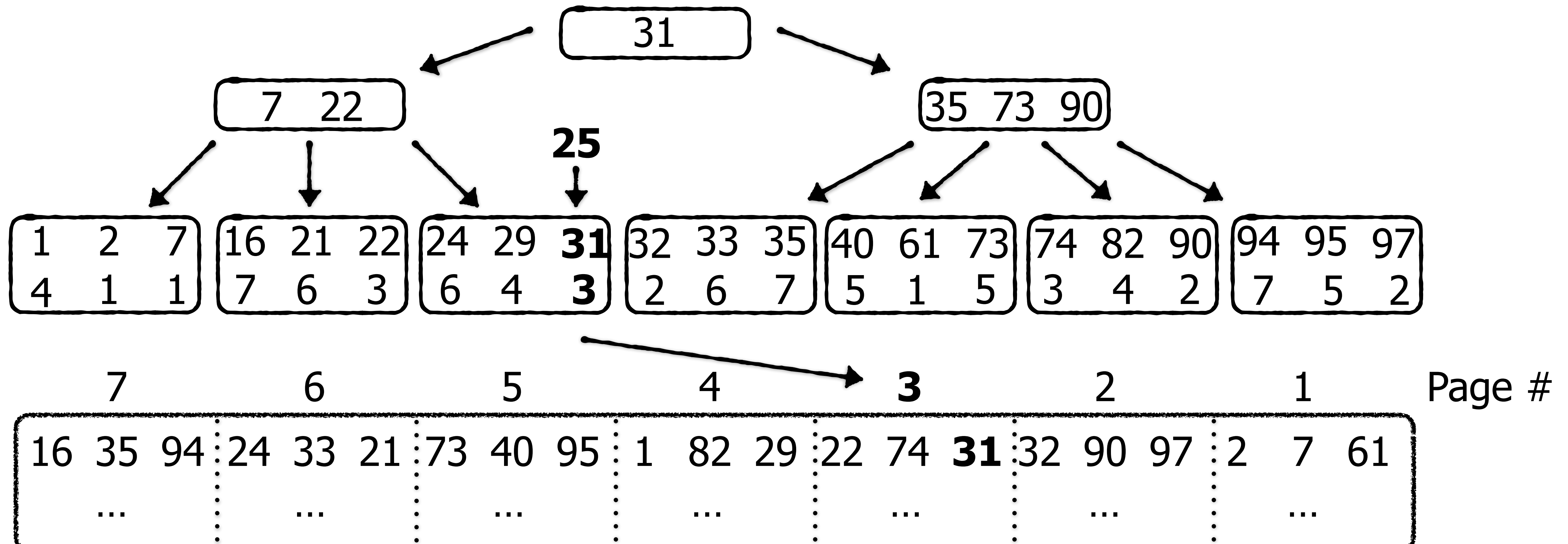
How do range scans work?

Select * from table where $A > 25$ and $A < 35$



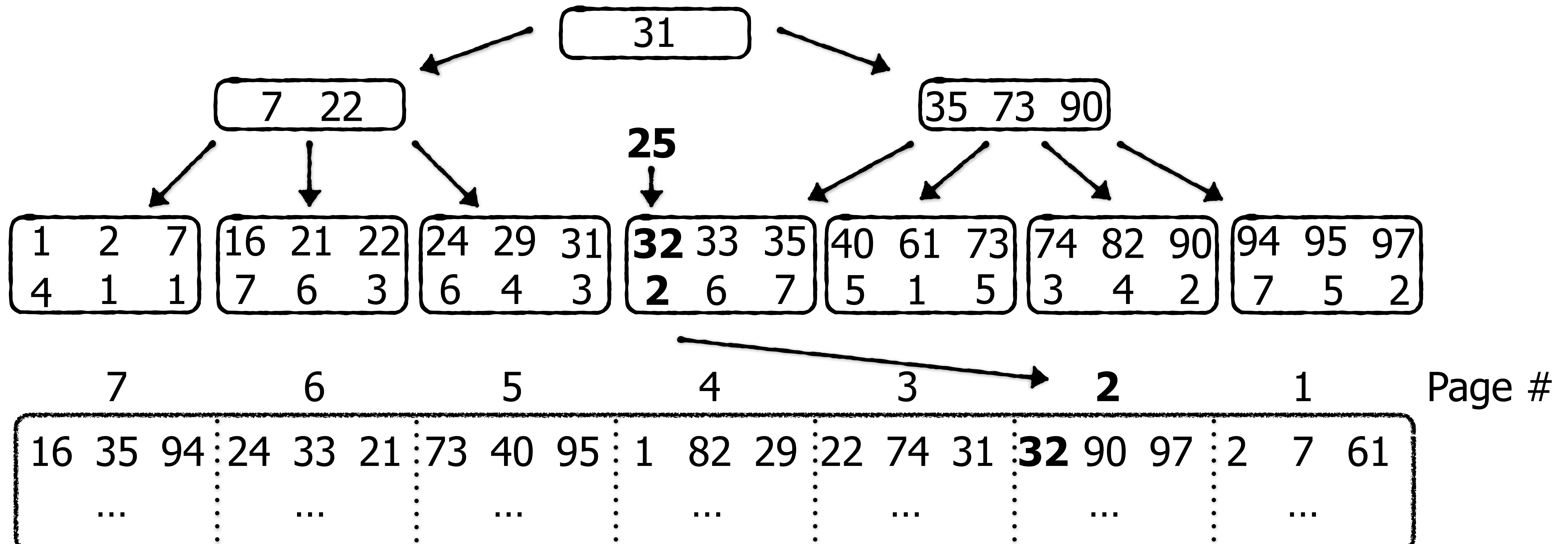
How do range scans work?

Select * from table where $A > 25$ and $A < 35$



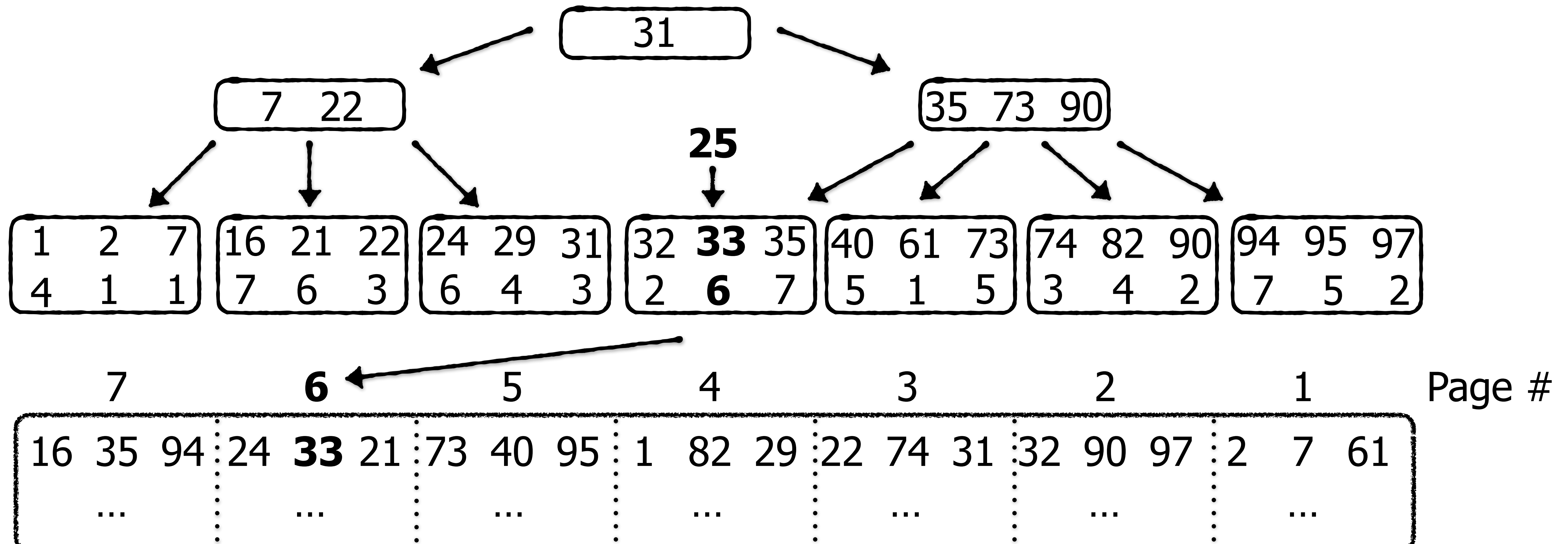
How do range scans work?

Select * from table where $A > 25$ and $A < 35$



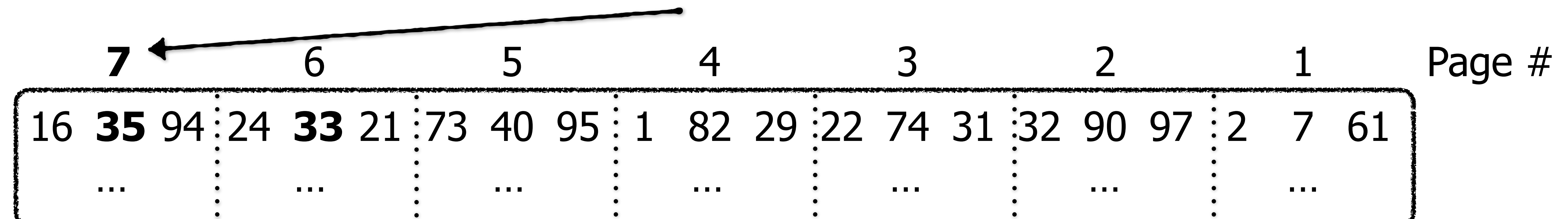
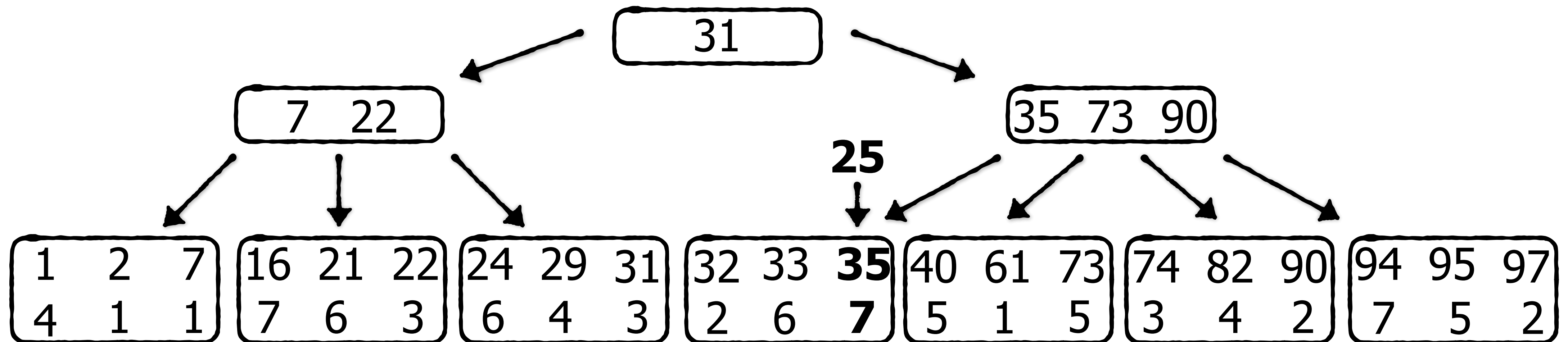
How do range scans work?

Select * from table where $A > 25$ and $A < 35$

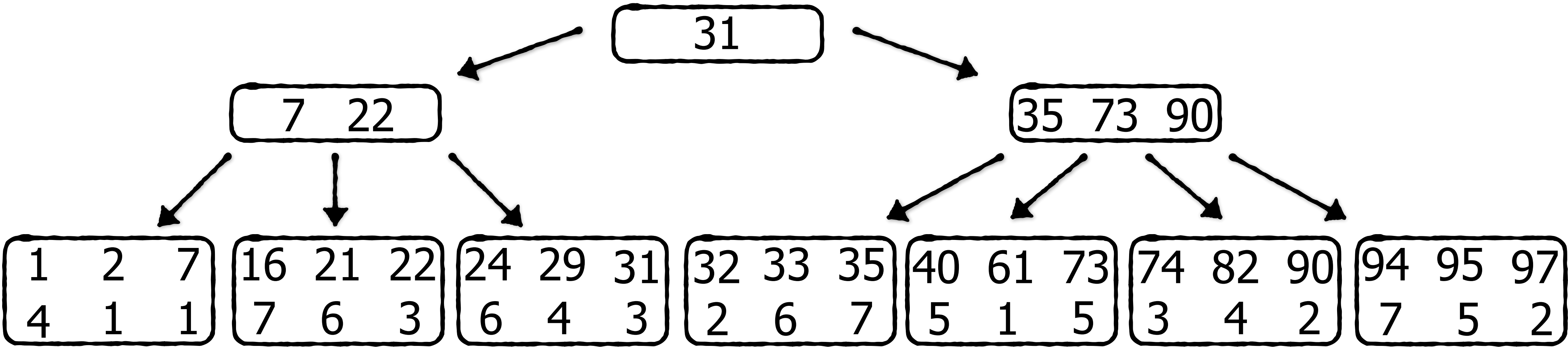


Scan Cost?

Select A, B from table where $A > x$ and $A < y$



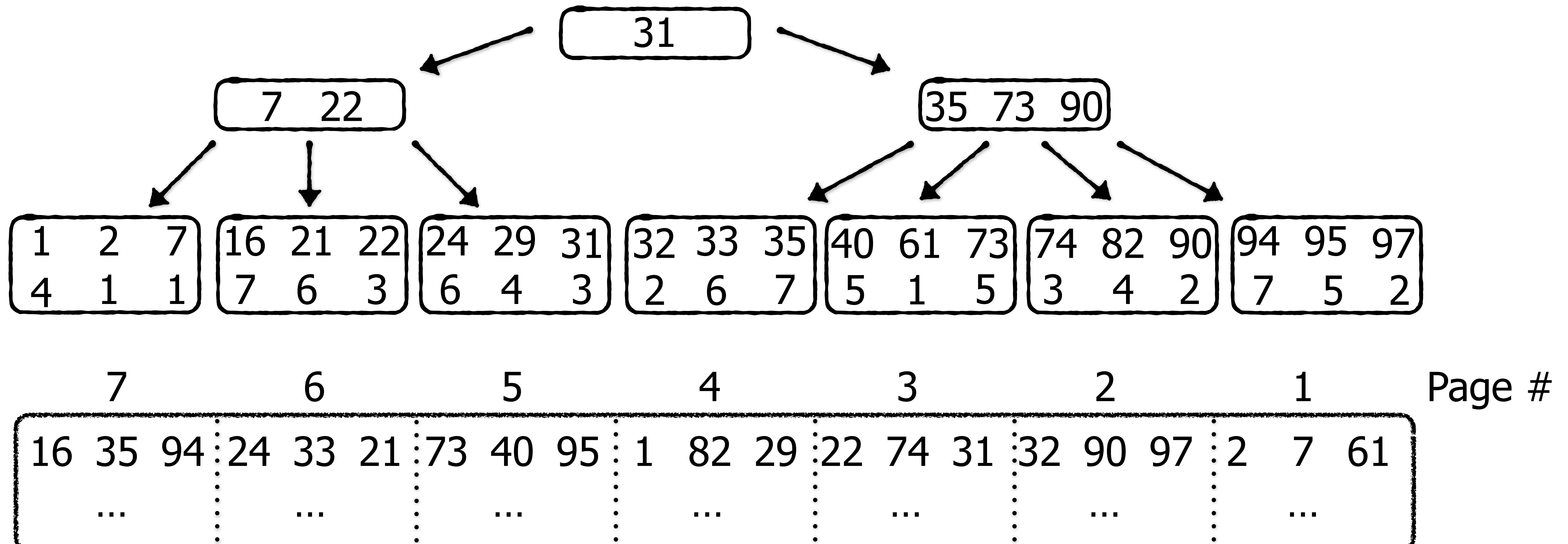
Scan Cost? $O(\log_B N + S)$ I/Os $S = \#$ entries in the range



7			6			5			4			3			2			1			Page #
16	35	94	24	33	21	73	40	95	1	82	29	22	74	31	32	90	97	2	7	61	
...			...			...			...			...			...			...			

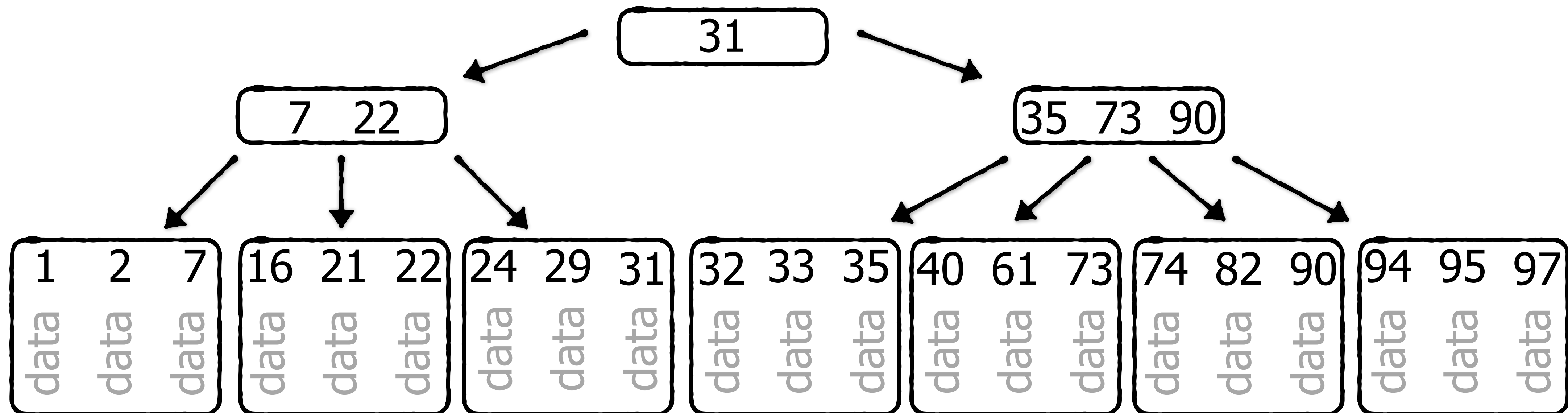
Scan Cost? $O(\log_B N + S)$ I/Os $S = \#$ entries in the range

Can we do better?

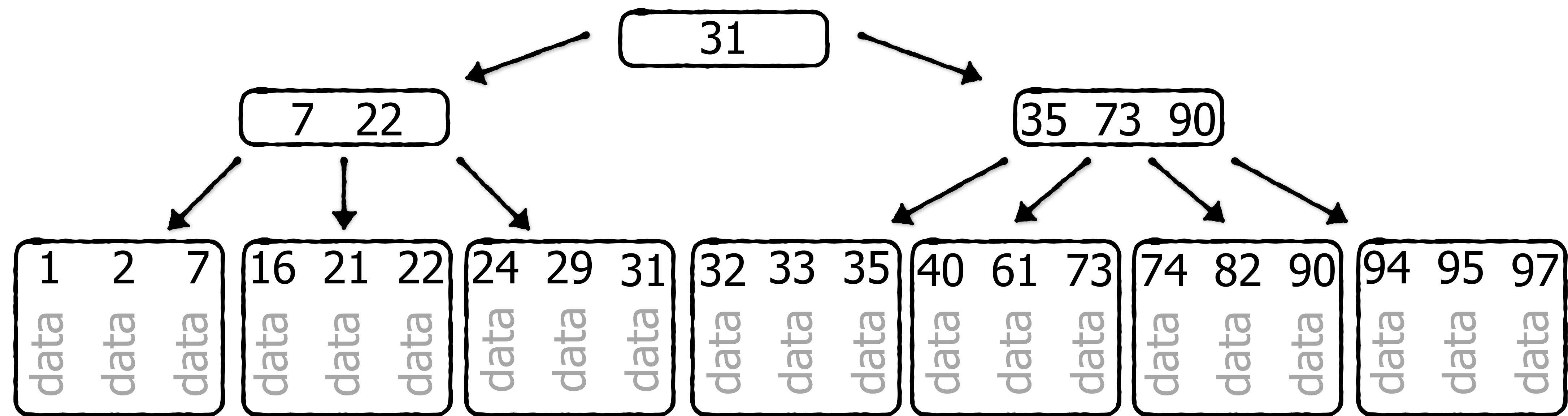


Scan Cost? $O(\log_B N + S)$ I/Os $S = \#$ entries in the range

Can we do better? **Yes, by storing the table's data within the index**

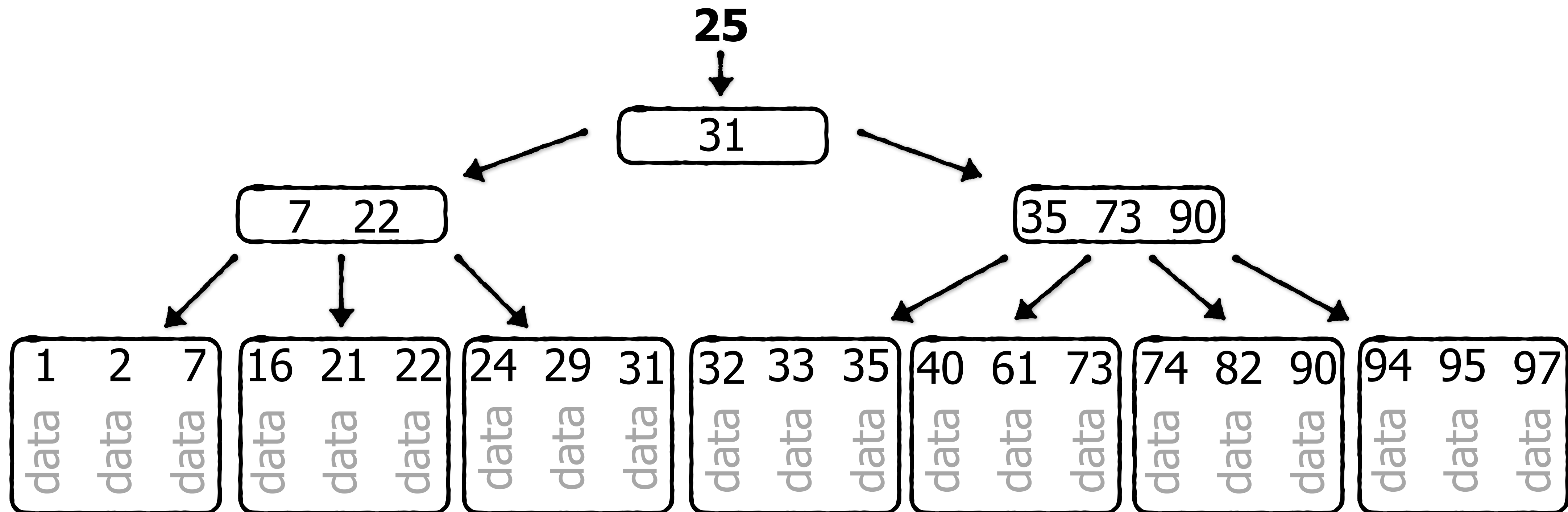


This is known as a “Clustered Index”



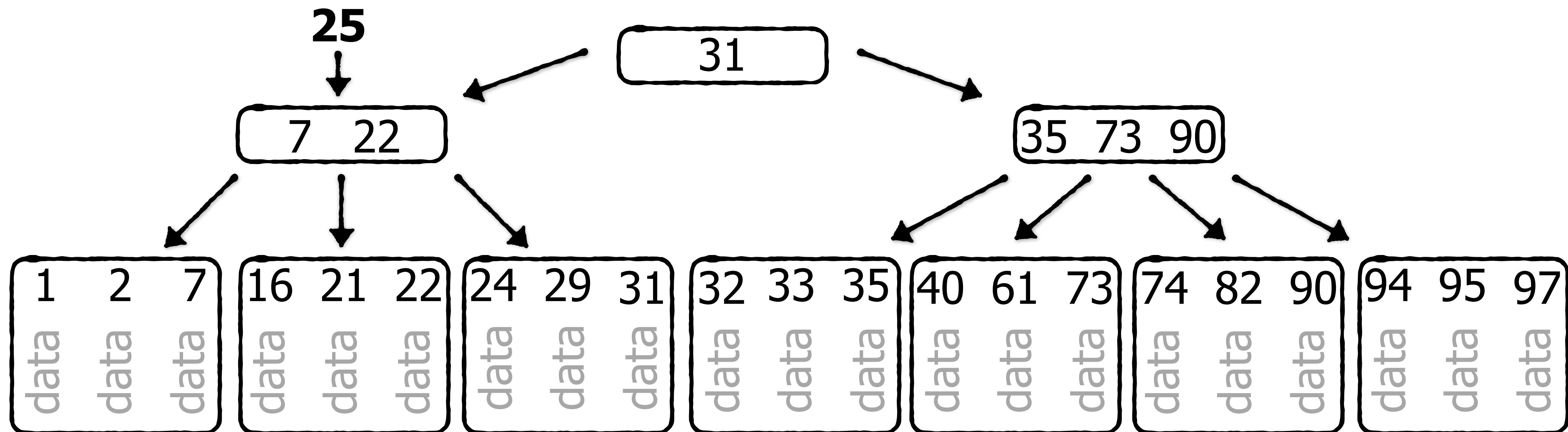
This is known as a "Clustered Index"

Let's try again Select * from table where A > 25 and A < 35



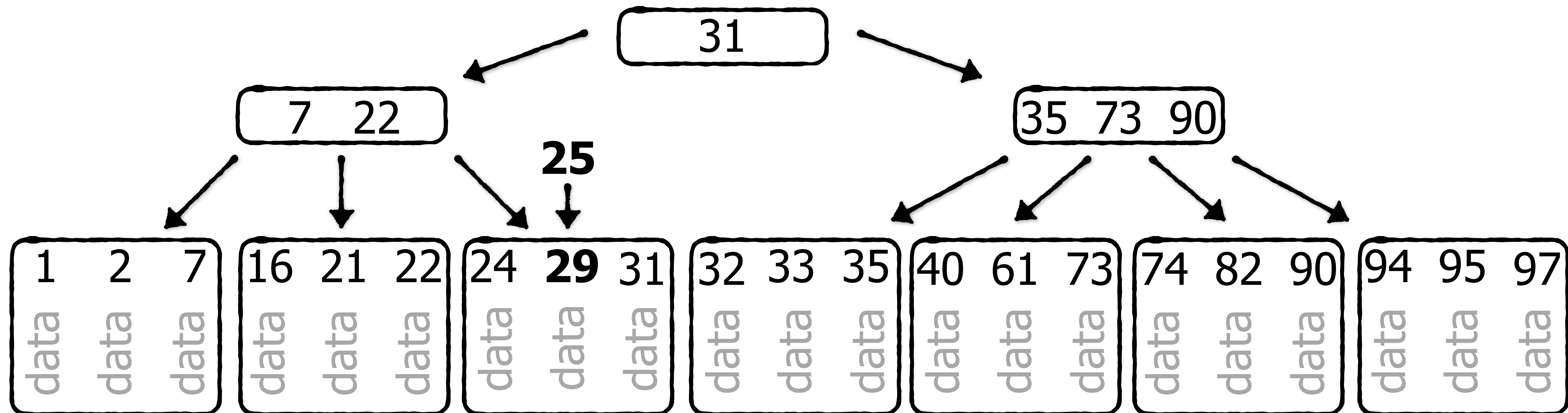
This is known as a "Clustered Index"

Let's try again Select * from table where A > 25 and A < 35



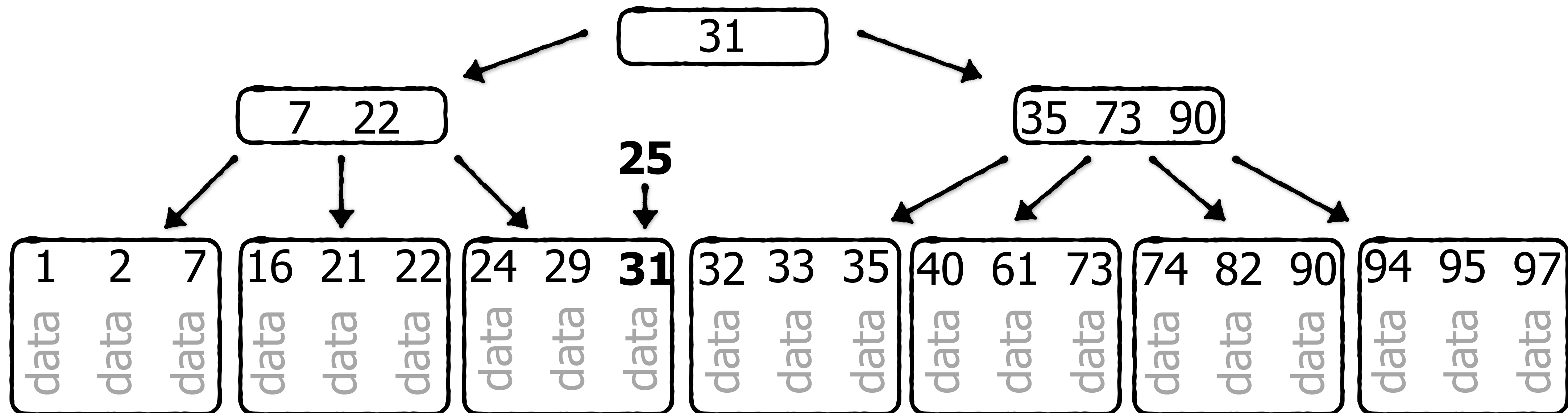
This is known as a "Clustered Index"

Let's try again Select * from table where A > 25 and A < 35



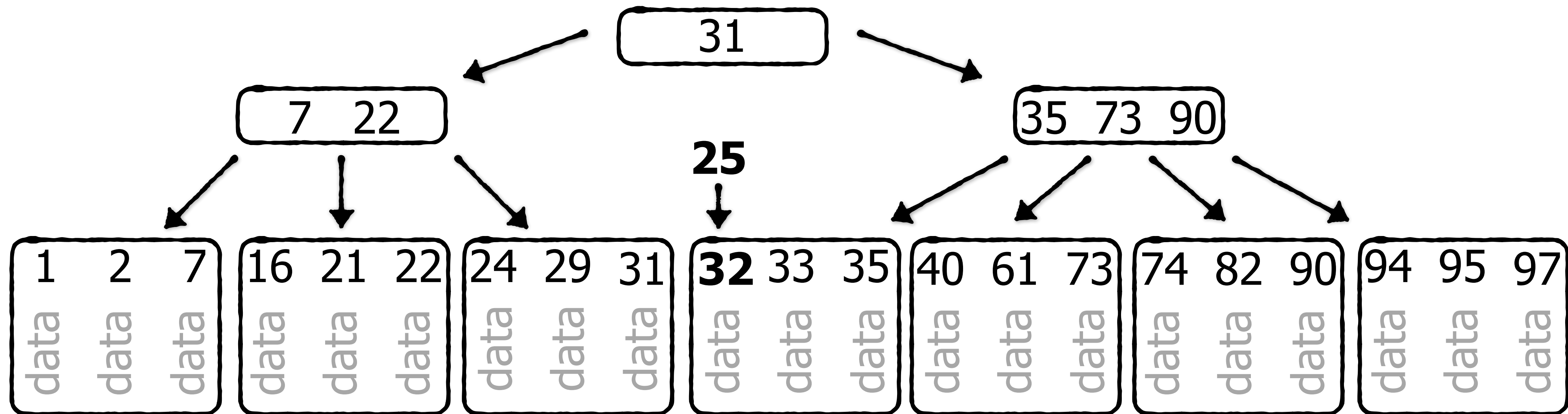
This is known as a "Clustered Index"

Let's try again Select * from table where A > 25 and A < 35



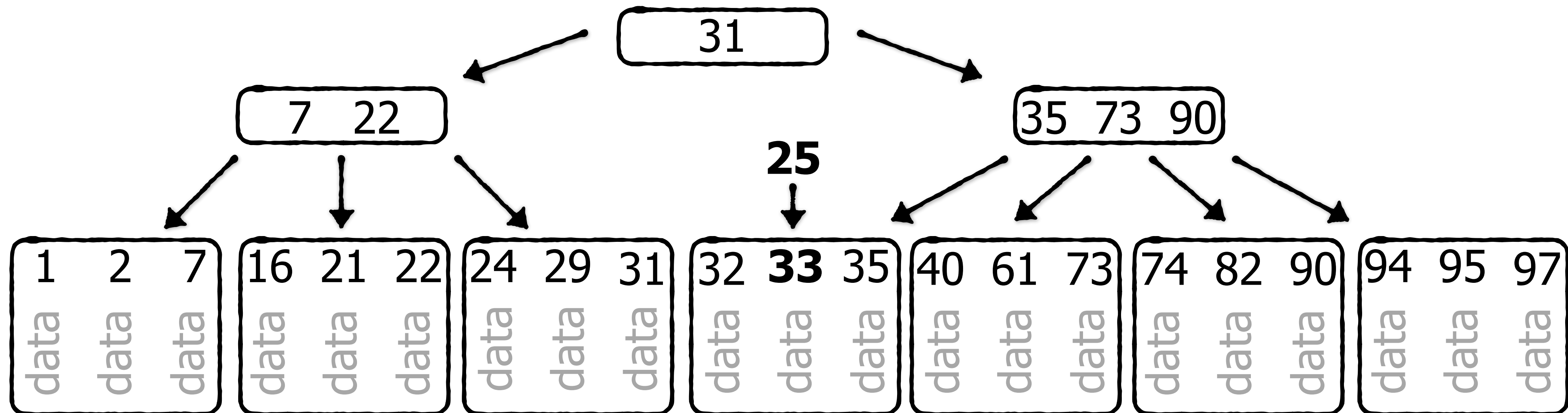
This is known as a "Clustered Index"

Let's try again Select * from table where A > 25 and A < 35



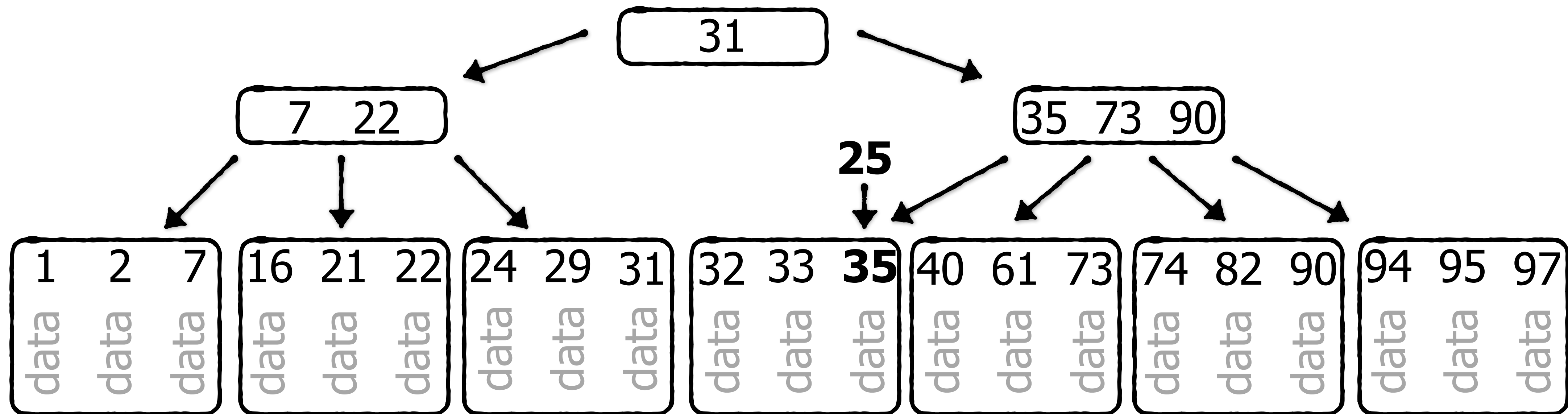
This is known as a "Clustered Index"

Let's try again Select * from table where A > 25 and A < 35



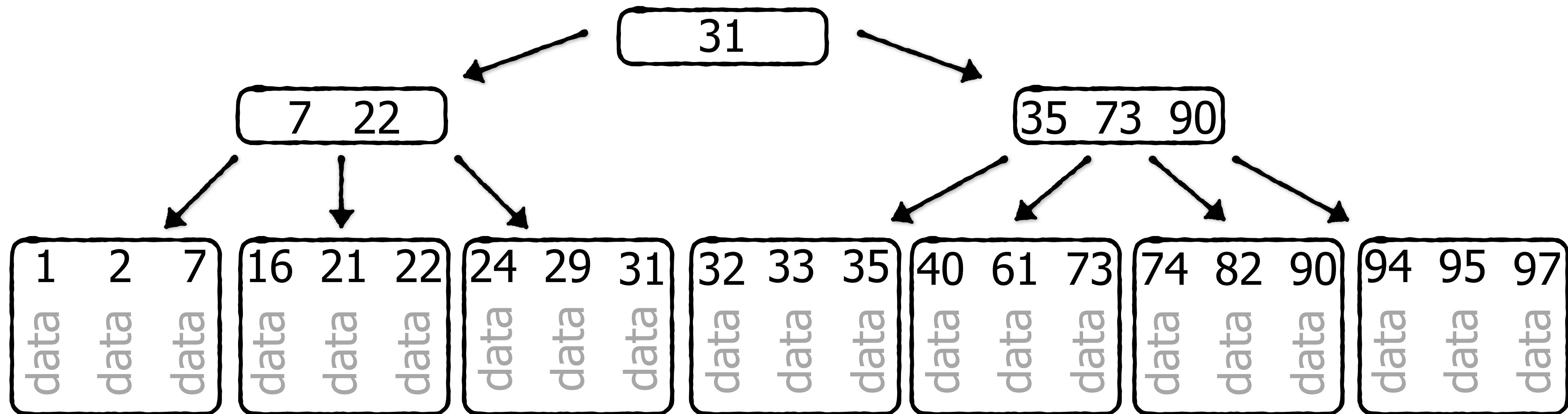
This is known as a "Clustered Index"

Let's try again Select * from table where A > 25 and A < 35



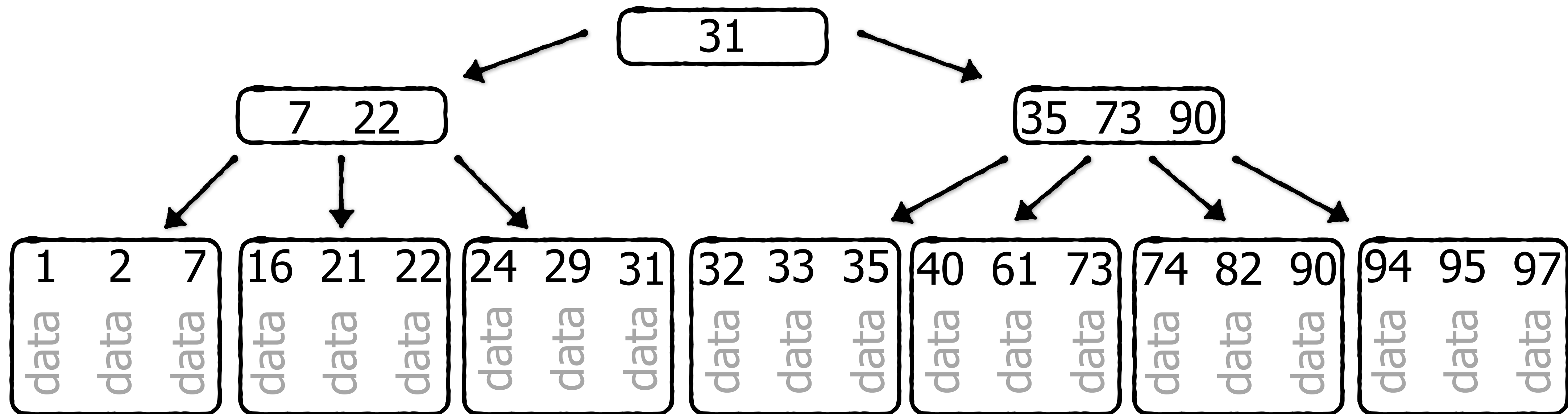
Unclustered Index cost model: $O(\log_B N + S)$

Clustered Index cost model?



Unclustered Index cost model: $O(\log_B N + S)$

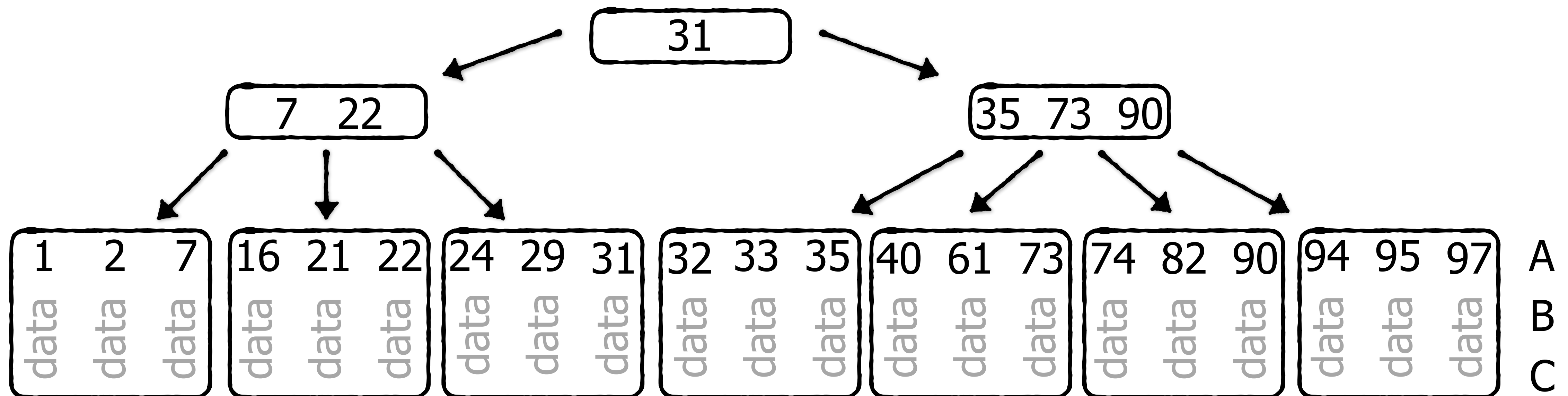
Clustered Index cost model? **$O(\log_B N + S/B)$**



Unclustered Index cost model: $O(\log_B N + S)$

Clustered Index cost model? **$O(\log_B N + S/B)$**

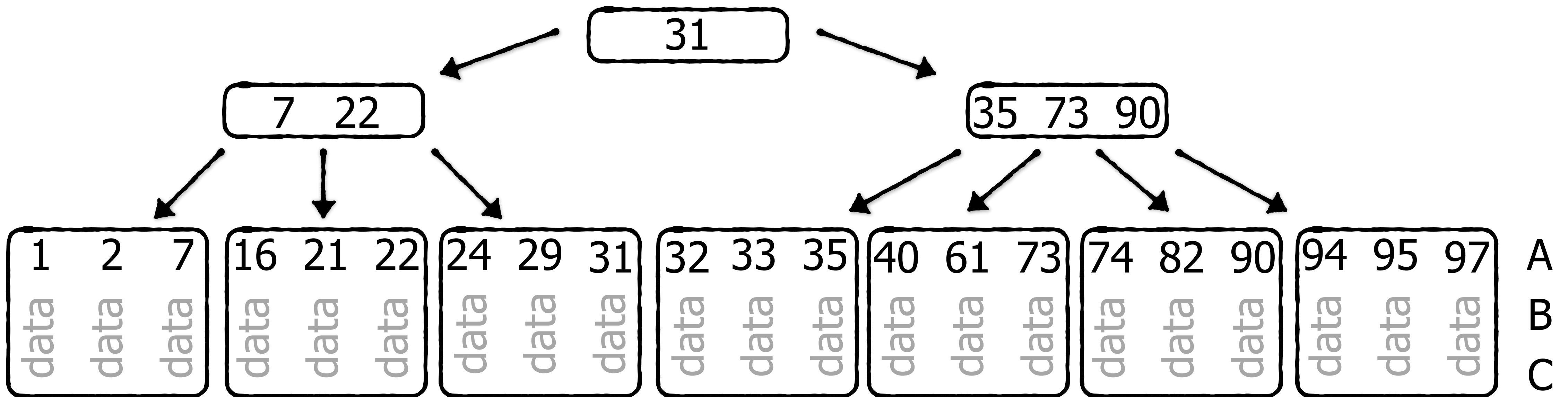
Shortcomings?



Unclustered Index cost model: $O(\log_B N + S)$

Clustered Index cost model? $O(\log_B N + S/B)$

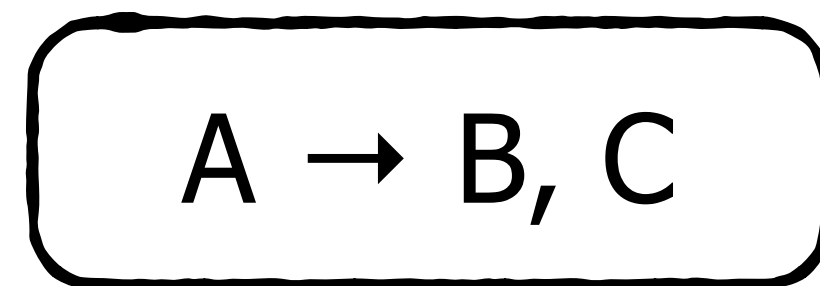
Shortcomings? Can only sort table based on one column at a time. So usually there is only 1 clustered index per table (Unless we are willing to duplicate the table)



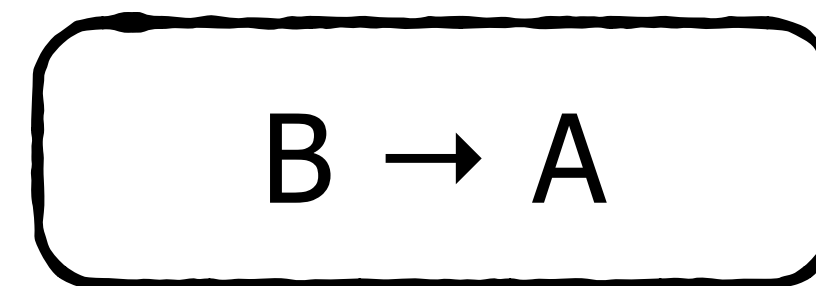
We can have multiple unclustered indexes in addition to one clustered index

For example:

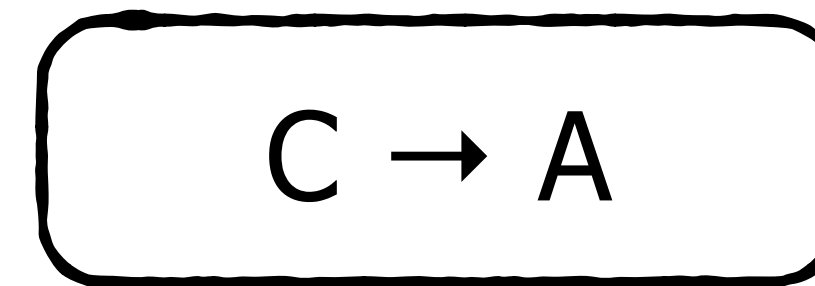
Clustered(A)



Unclustered(B)



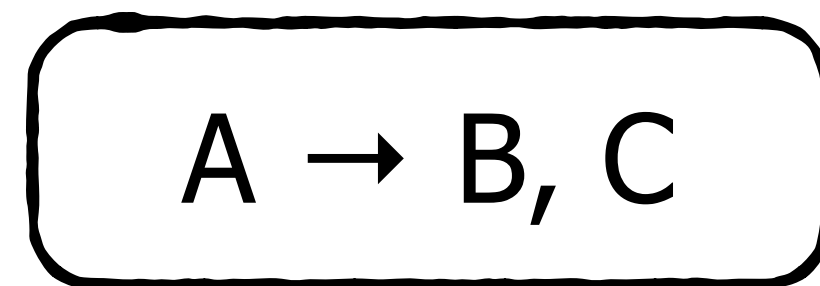
Unclustered(C)



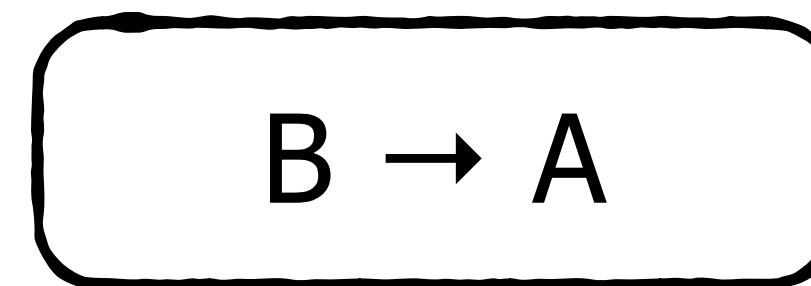
We can have multiple unclustered indexes in addition to one clustered index

For example:

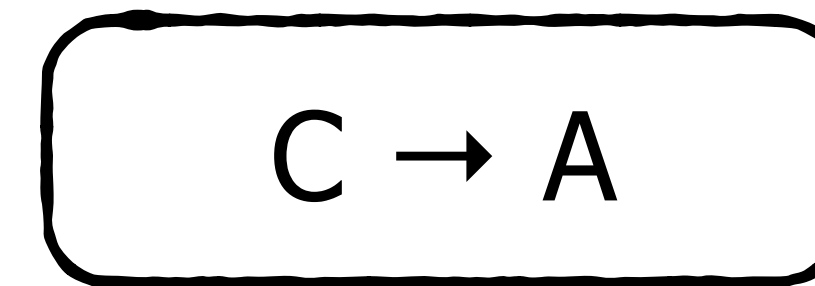
Clustered(A)



Unclustered(B)



Unclustered(C)



allows optimizing multiple queries filtering on different columns.

A B-tree insertion/update/delete costs at least $O(1)$ write I/O

For write-heavy workloads, this is pricey. Can we reduce this more towards $O(1/B)$?

Next week: the Log-Structured Merge-Trees