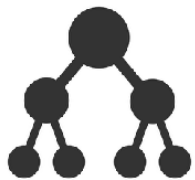
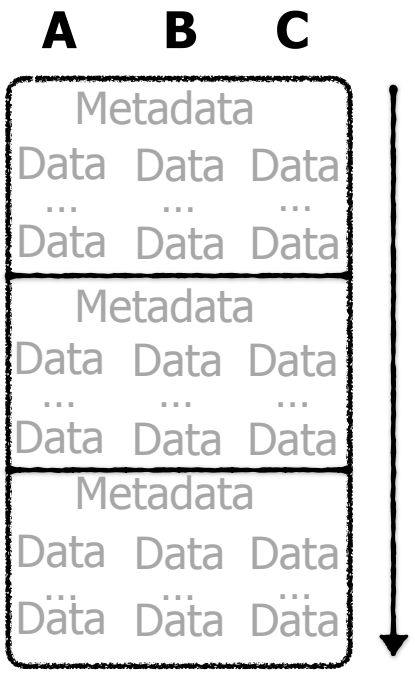
Indexing



Database System Technology

Niv Dayan



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

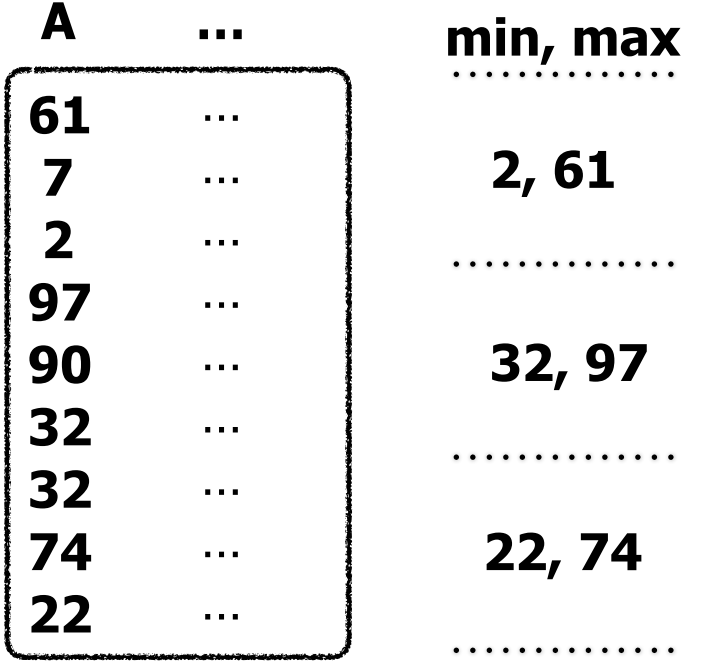
(3) Binary tree

(4) Hash table

(5) Extendible Hashing

(6) B-tree

(1) Zone Maps



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

(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

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
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

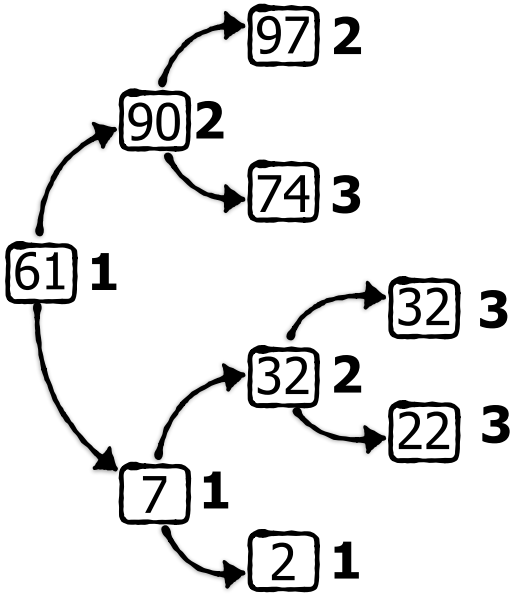
A	B	C
2	b3	c3
7	b2	c2
22	b9	c9
32	b6	c6
32	b7	c7
50	b10	c10
61	b1	c1
74	b8	c8
90	b5	c5
97	b4	c4

Can you spot another disadvantage?

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

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

(3) Binary Tree



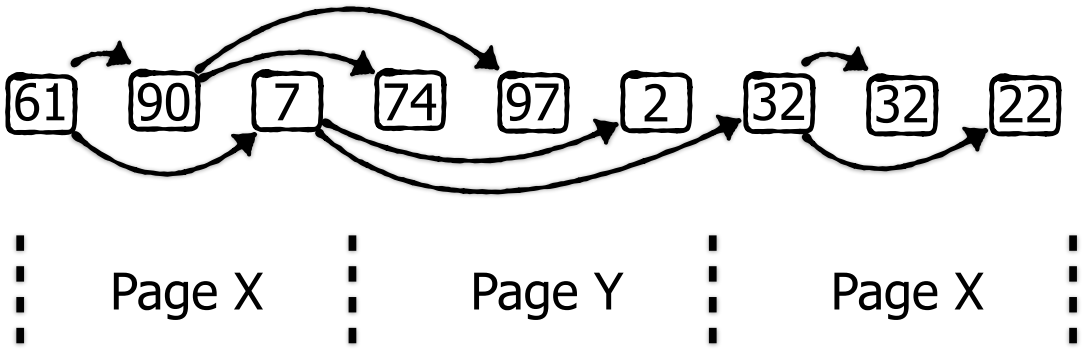
A	B	C
61	b1	c1
7	b2	c2
2	b3	c3
97	b4	c4
90	b5	c5
32	b6	c6
32	b7	c7
74	b8	c8
22	b9	c9

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

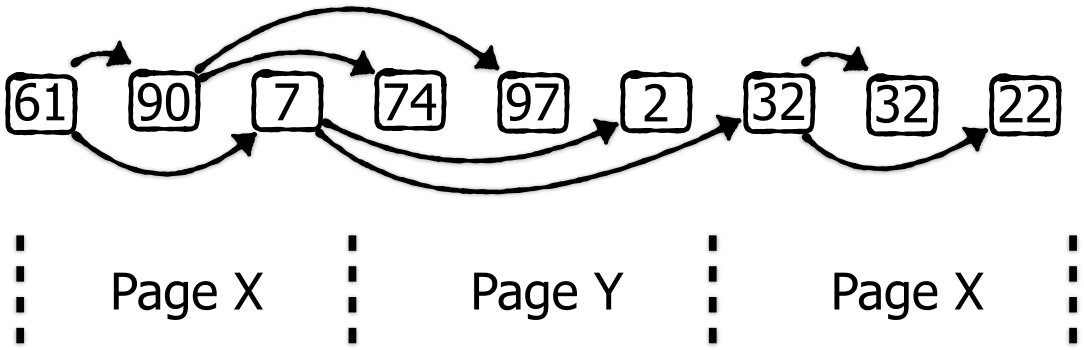
For example, it's possible to store a binary tree as an array and map it to different storage pages. (Assume same width keys for simplicity)



(3) Binary Tree

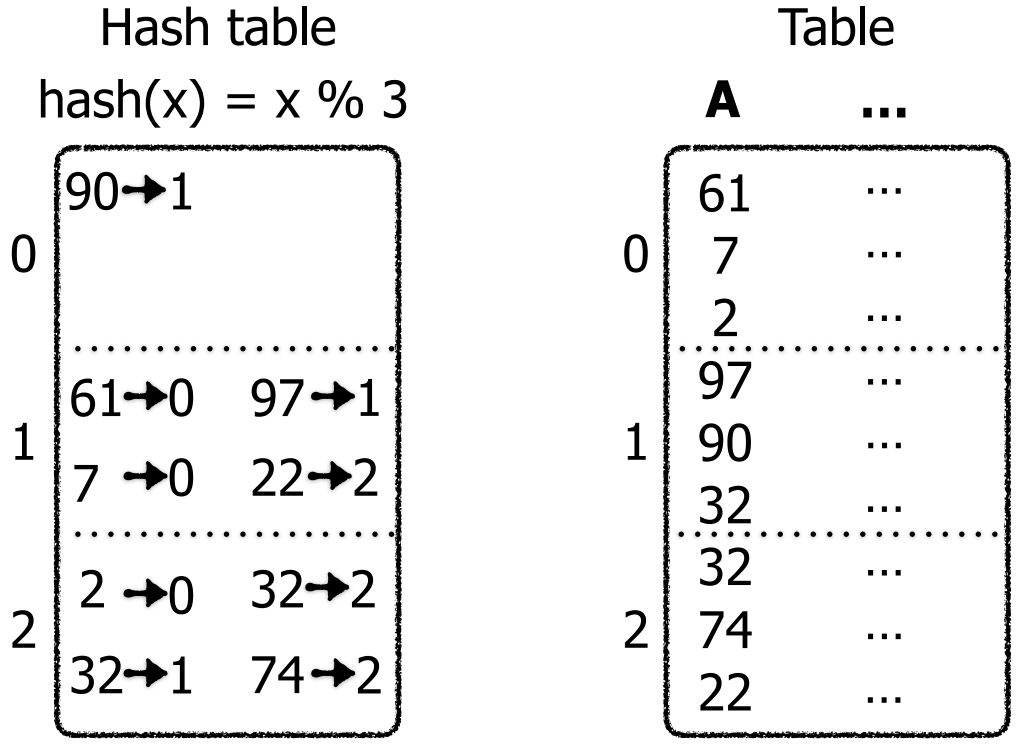
CPU cost: **$O(\log_2(N))$**

I/O cost: **$O(\log_2(N/B))$** Since the first $O(\log_2(B))$ levels fit in first page



(4) Hash Table - in storage

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



(4) Hash Table - in storage

Read/write I/O cost:

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

Downsides?

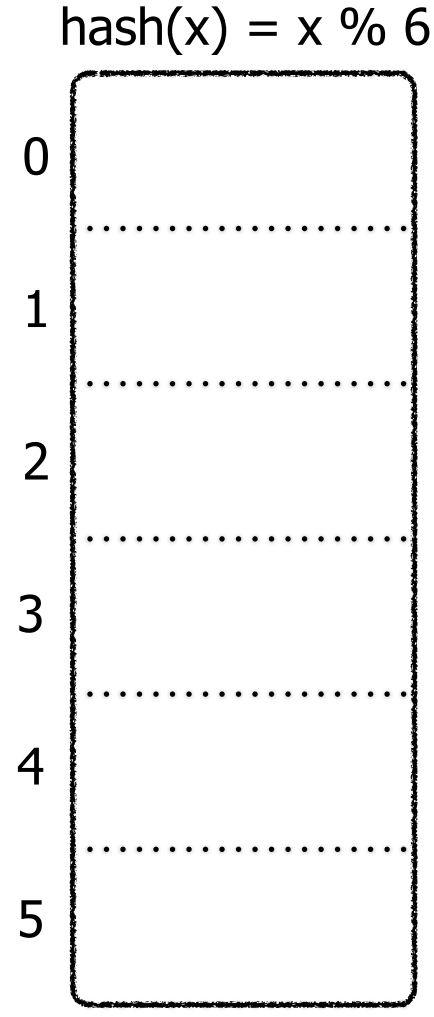
(1) No support for range reads

(2) Expansion leads to performance slumps

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

↑

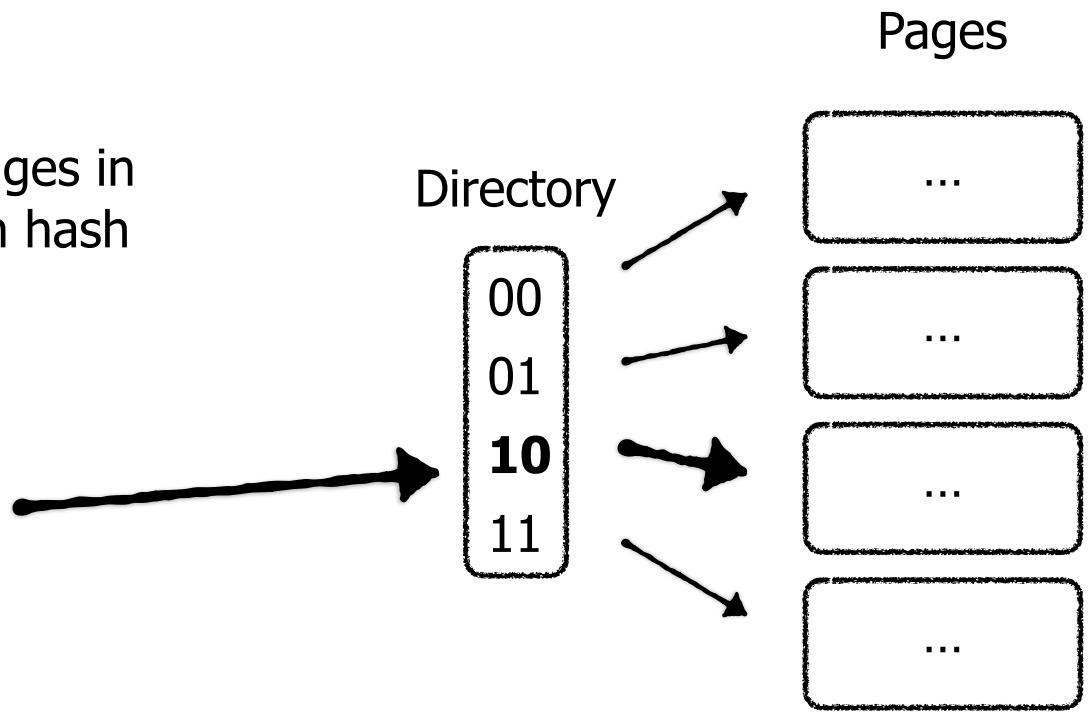
Let's address points (2) and (3)



(5) Extendible Hashing

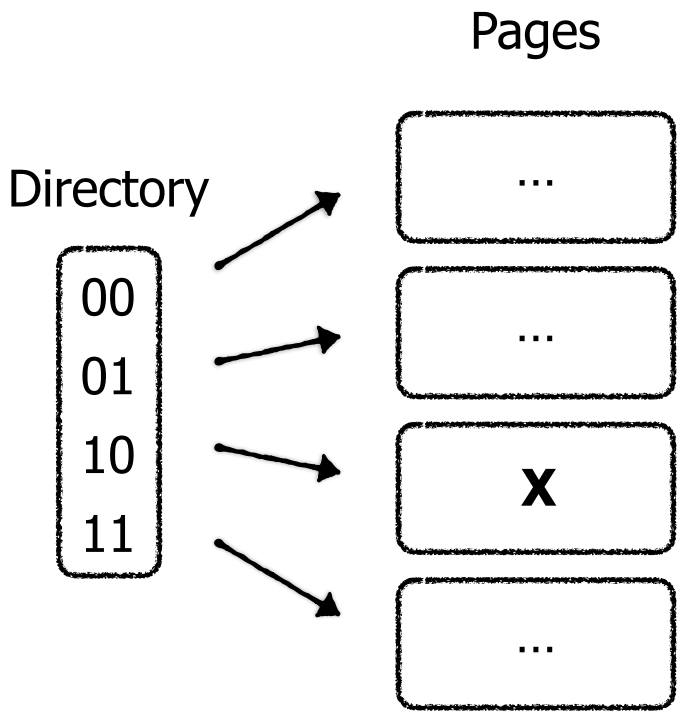
A directory maps pages in storage with a given hash prefix

e.g., insert key X
Hash(X) = 10…



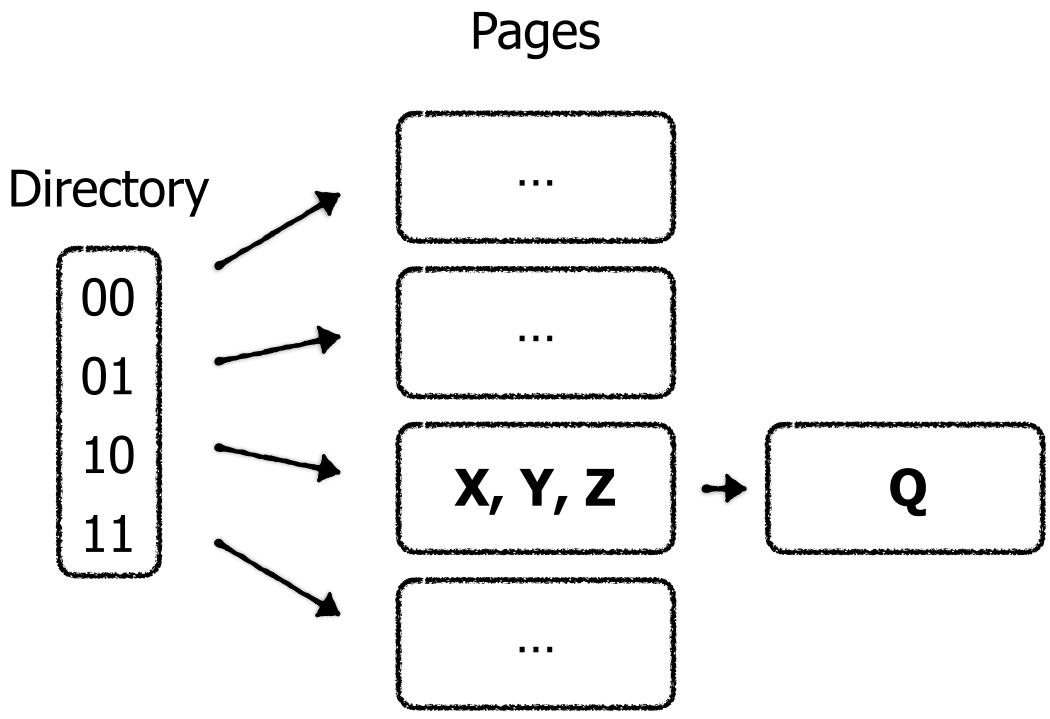
(5) Extendible Hashing

A directory maps pages in storage with a given hash prefix



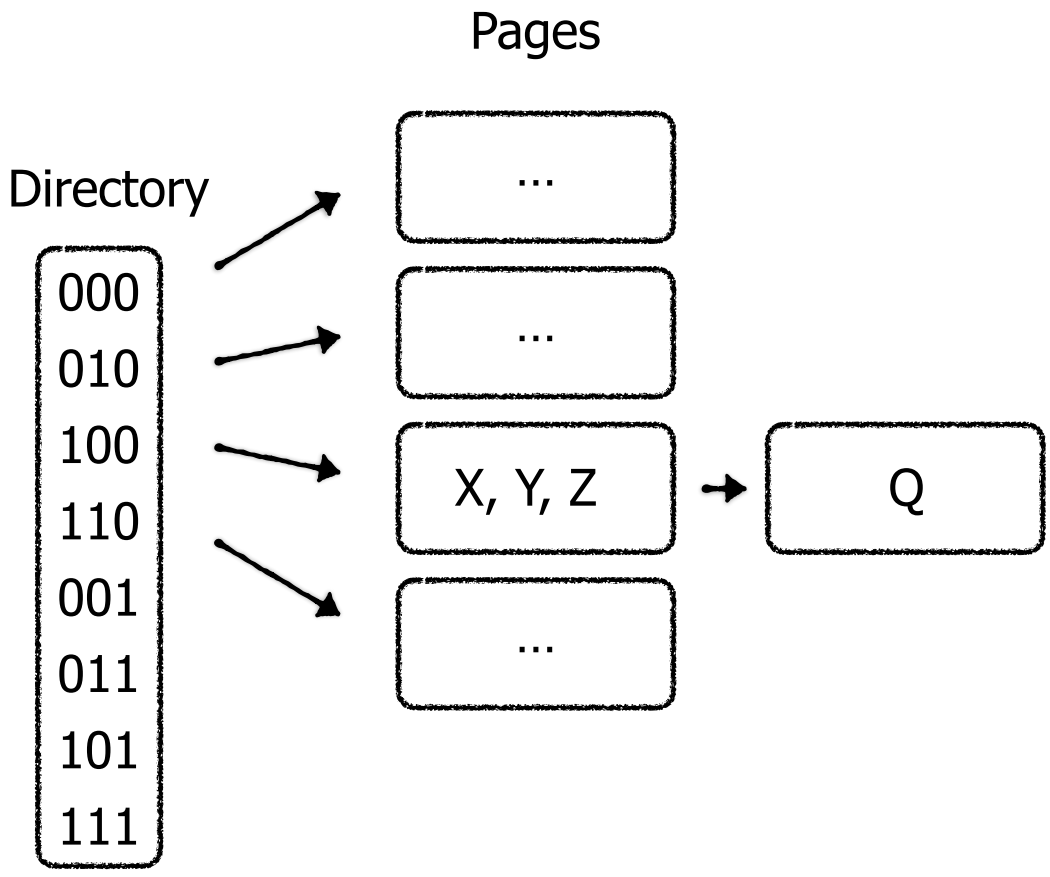
(5) Extendible Hashing

Handle overflows via chaining



(5) Extendible Hashing

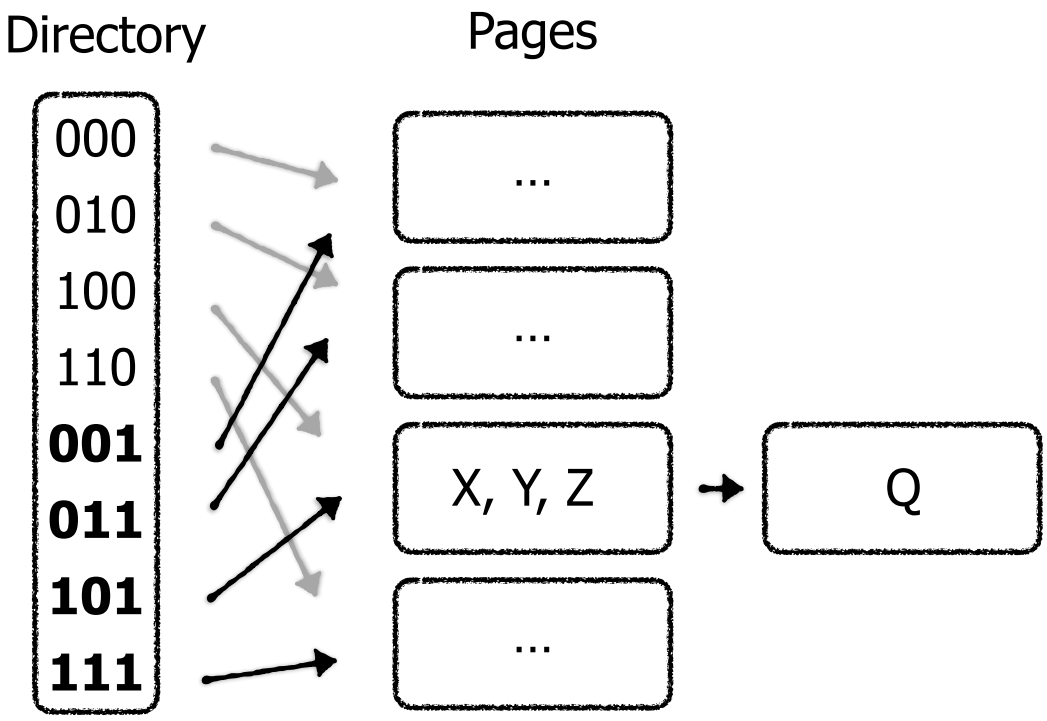
When we reach capacity, double directory size.



(5) Extendible Hashing

When we reach capacity, double directory size.

New directory slots still point to previous pages



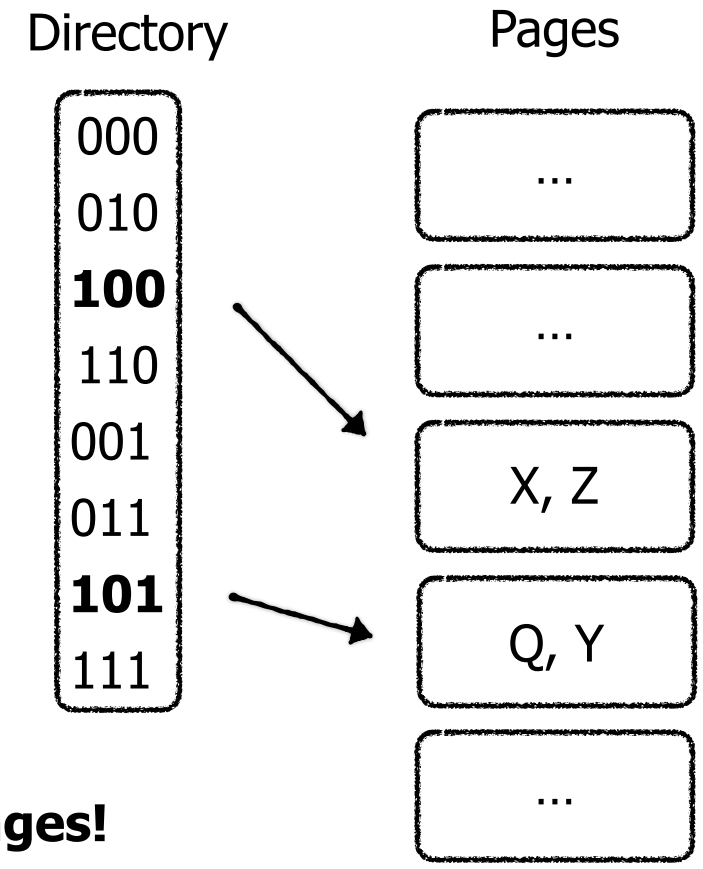
(5) Extendible Hashing

Comparison to Hash Table

(1) Lessor performance slumps due to expansion

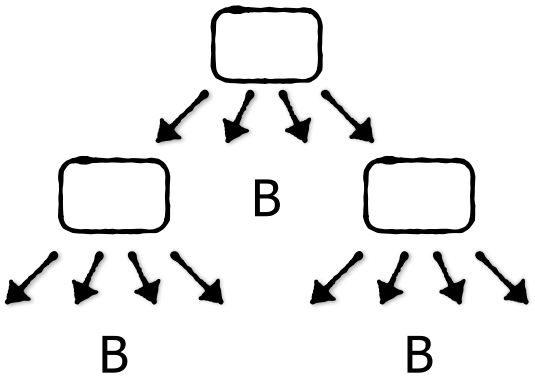
(2) Avoid wasting 50% space after expansion

(3) Directory consumes memory if it is cached, or it requires one extra I/O per operation if its in storage.



This is nice, but we still want to support ranges!

(6) B-Tree



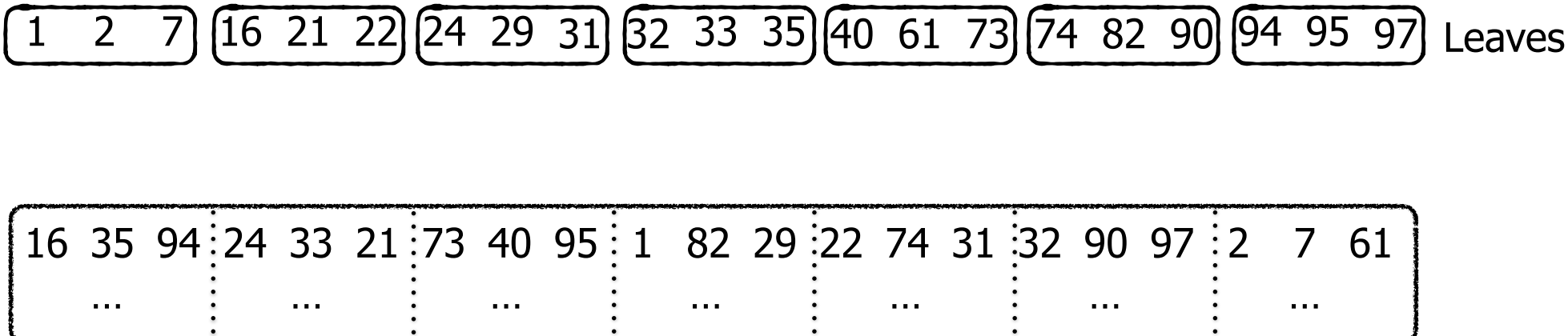
Invented in 1970 by Rudolph Bayer

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.

Proposal 5: B-Tree

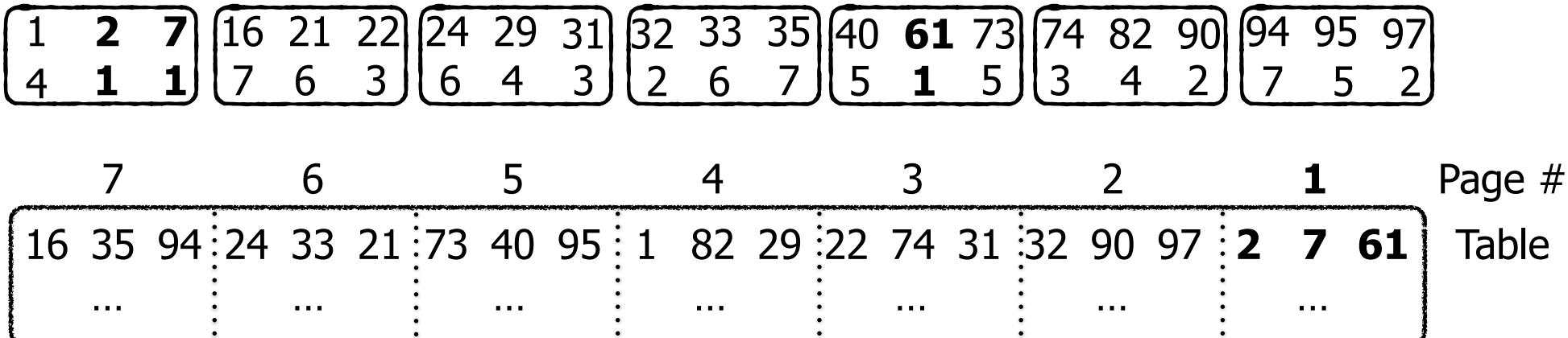
In this example, let's assume B=3, meaning each node contains 3 keys.



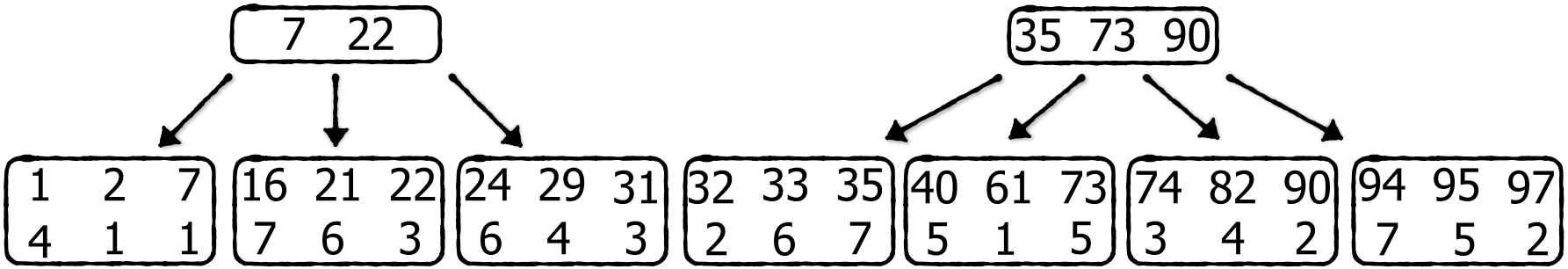
Proposal 5: B-Tree

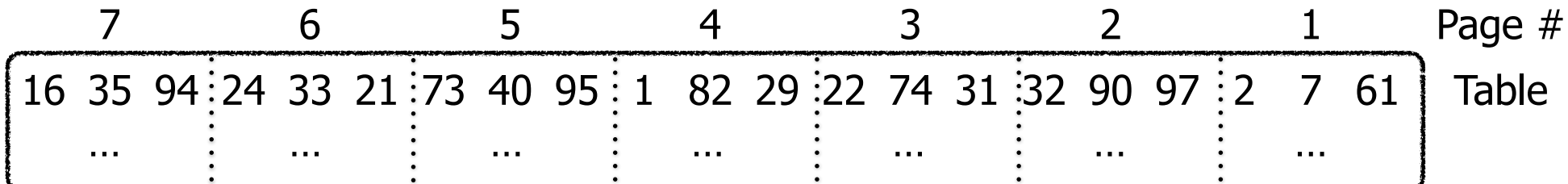
In this example, let's assume B=3, meaning each node contains 3 keys.

From each key, point to page the entry resides in



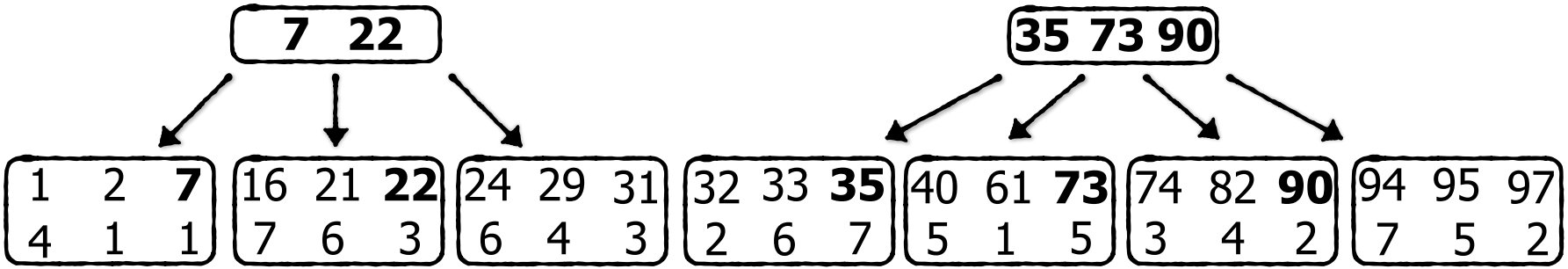
The next layer is of internal nodes, containing a delimiting key to separate between each pair of adjacent leaves.

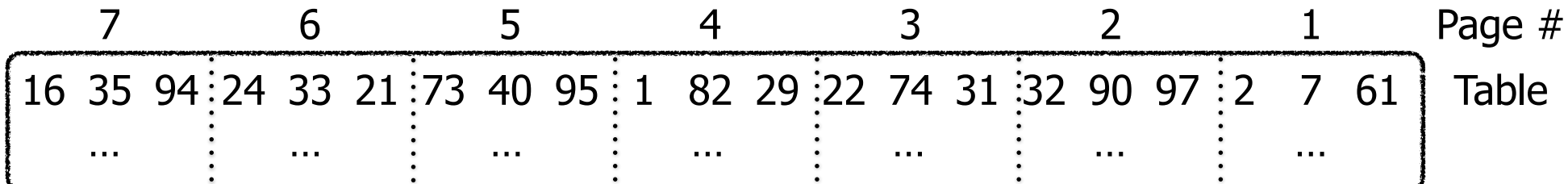




The next layer is of internal nodes, containing a delimiting key to separate between each pair of adjacent leaves.

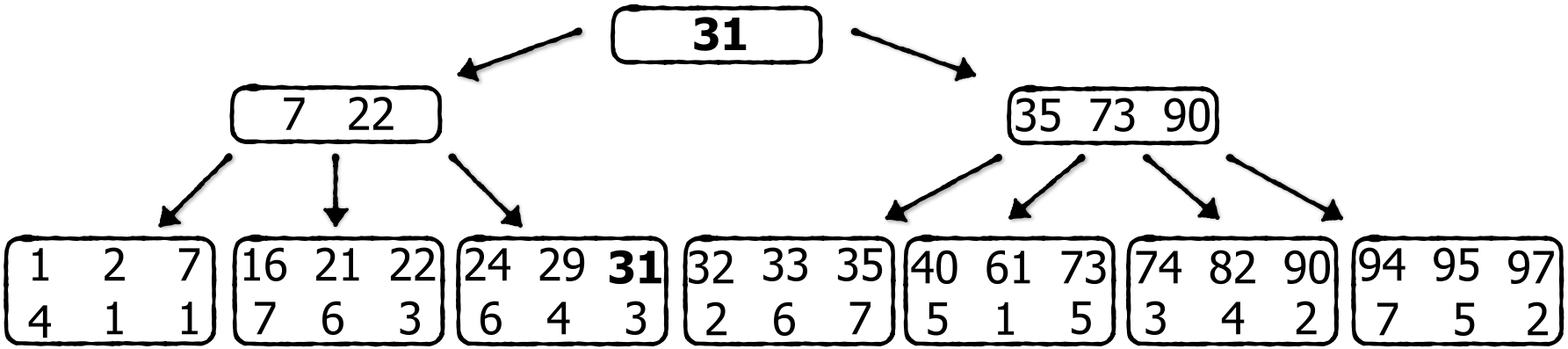
In this case, the delimiter is the largest key in sub-tree (except the last).

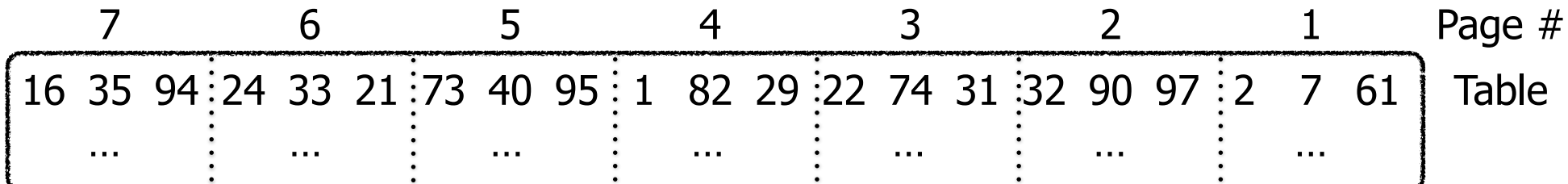




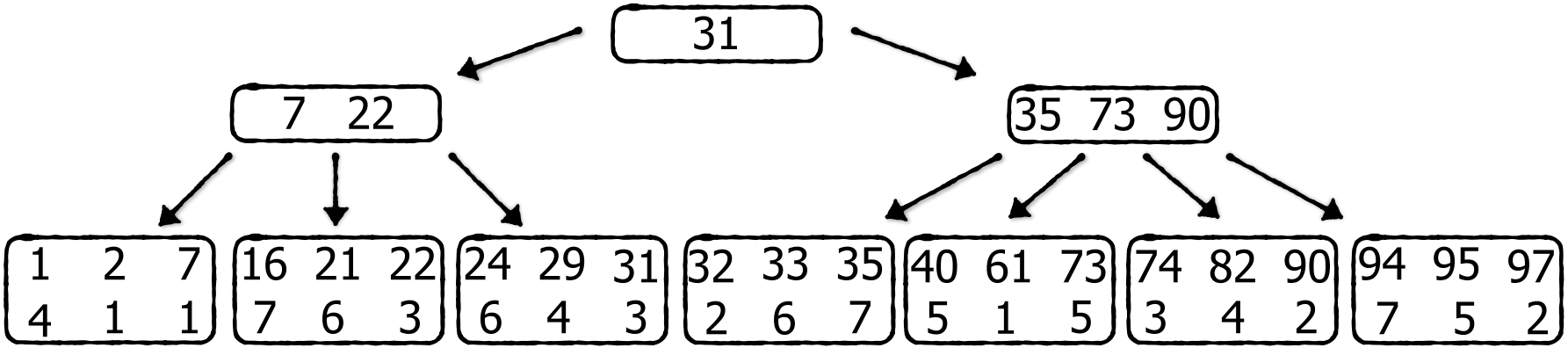
The next layer is of internal nodes, containing a delimiting key to separate between each pair of adjacent leaves.

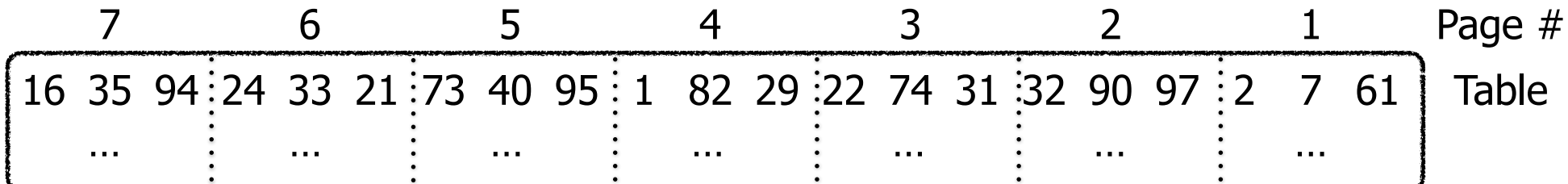
Apply this idea recursively.





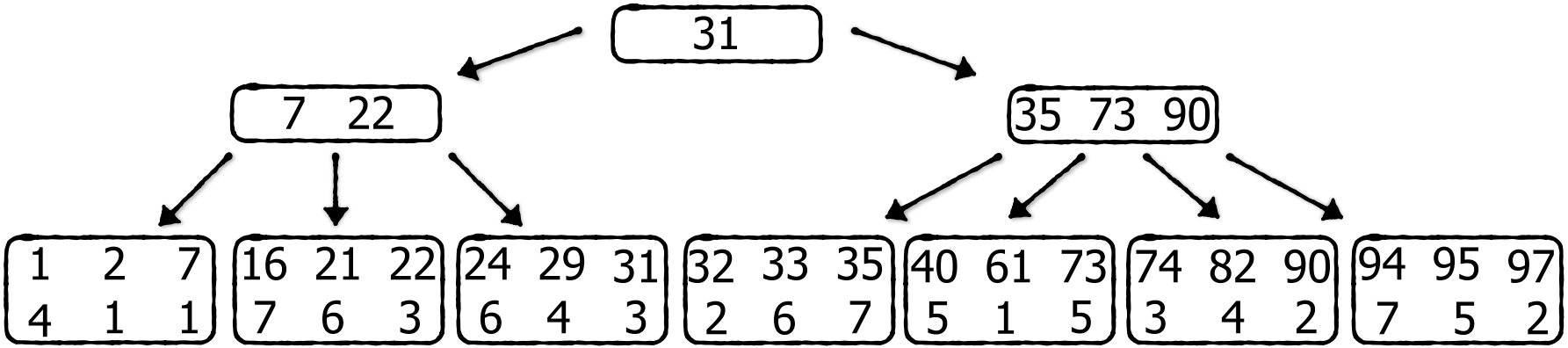
How to search?

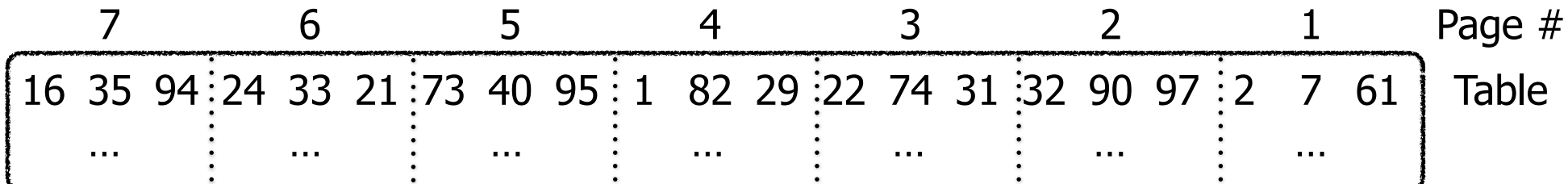




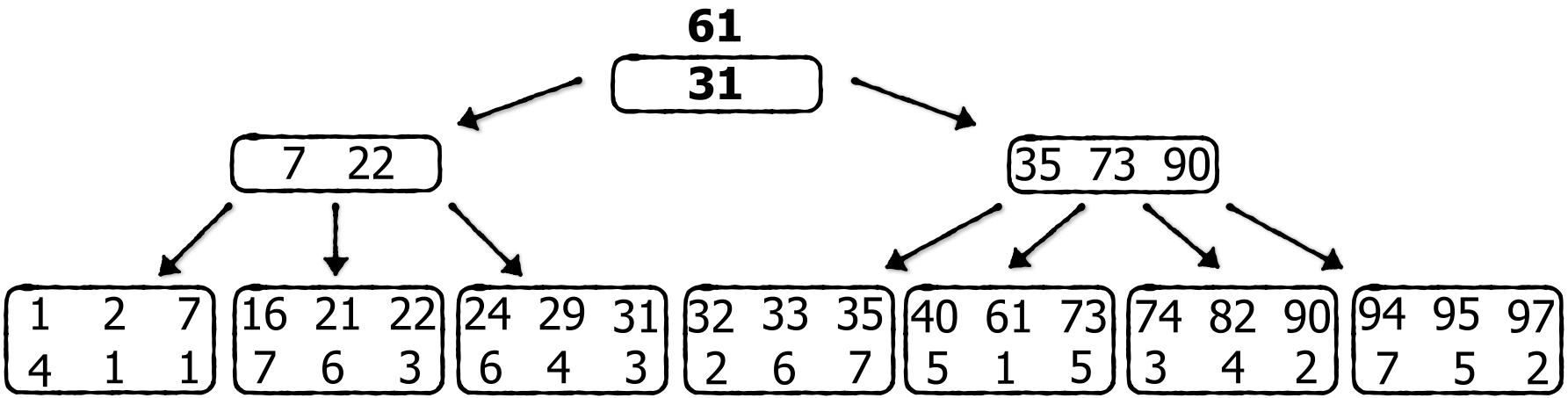
How to search? Binary search each node from root and follow matching pointer

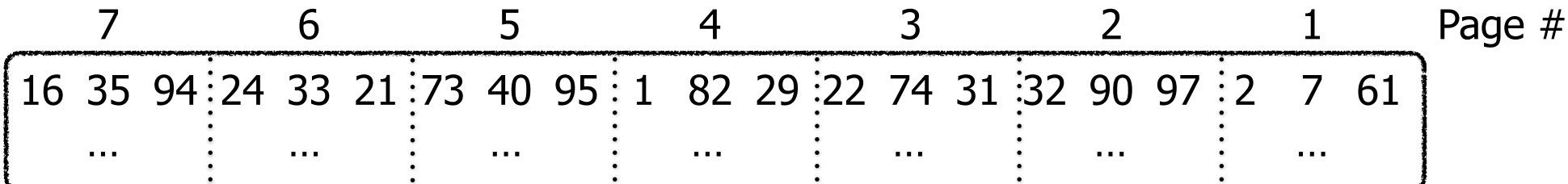
For example, search for **61**



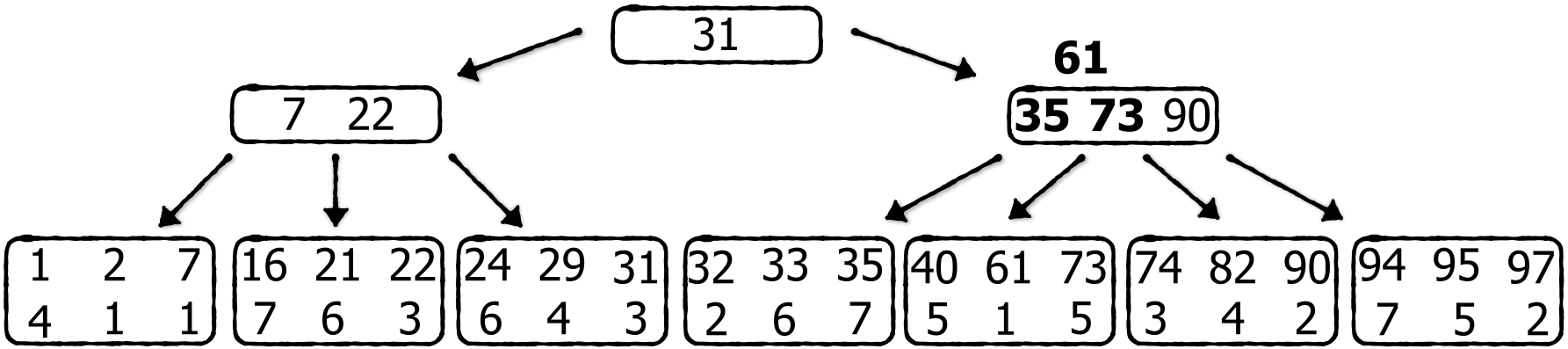


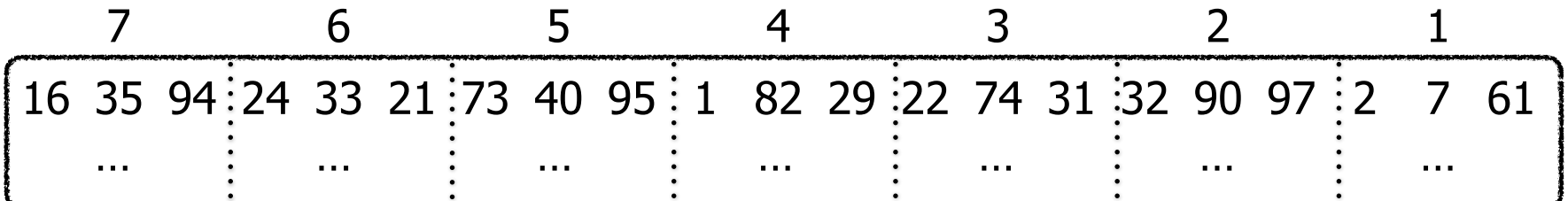
How to search? Binary search each node from root and follow matching pointer



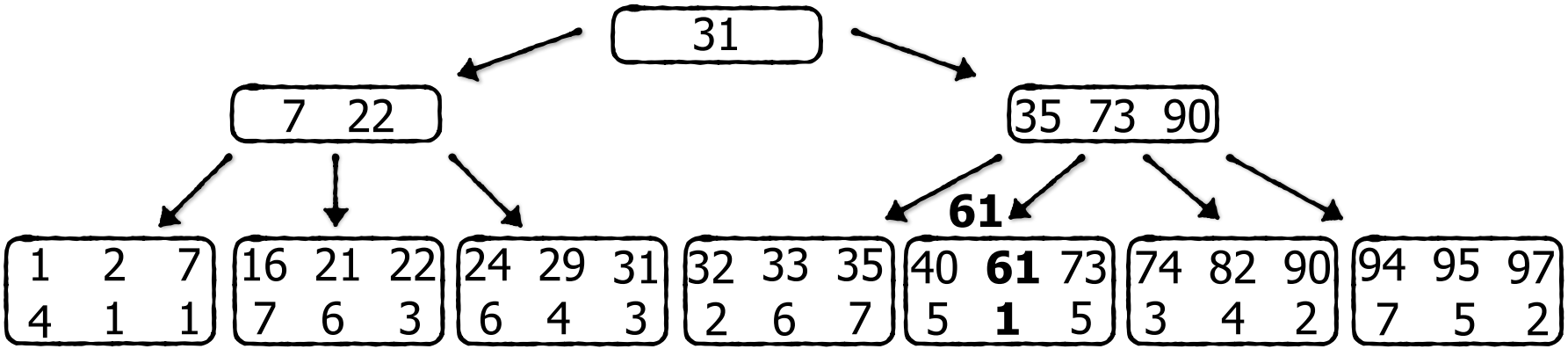


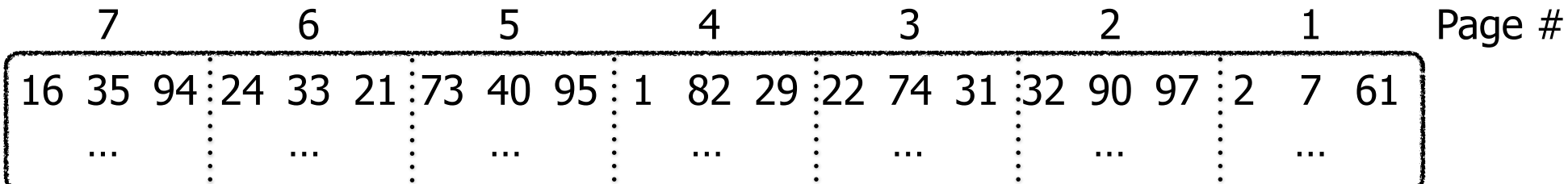
How to search? Binary search each node from root and follow matching pointer



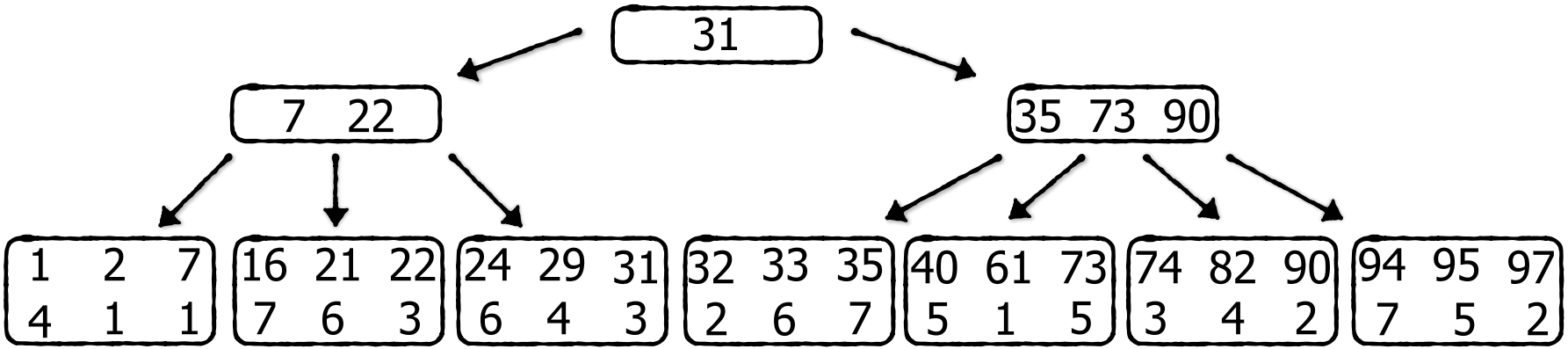


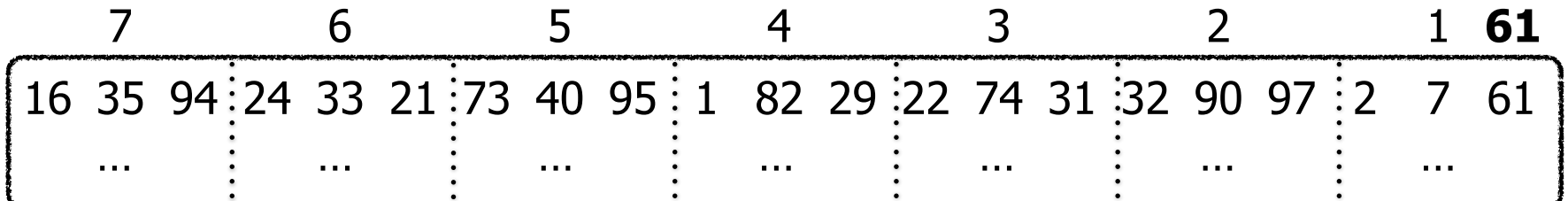
How to search? Binary search each node from root and follow matching pointer



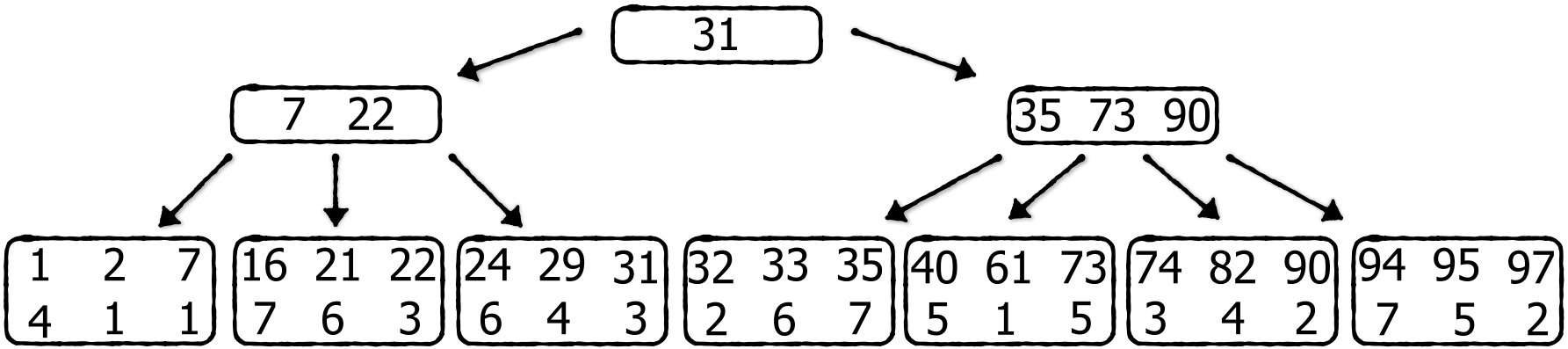


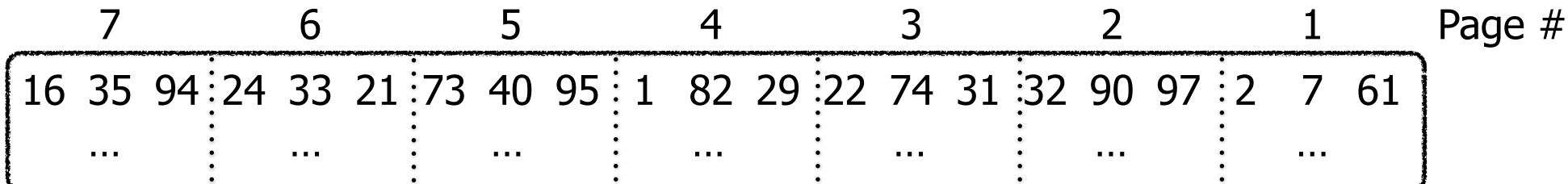
How to search? Binary search each node from root and follow matching pointer





Search cost?

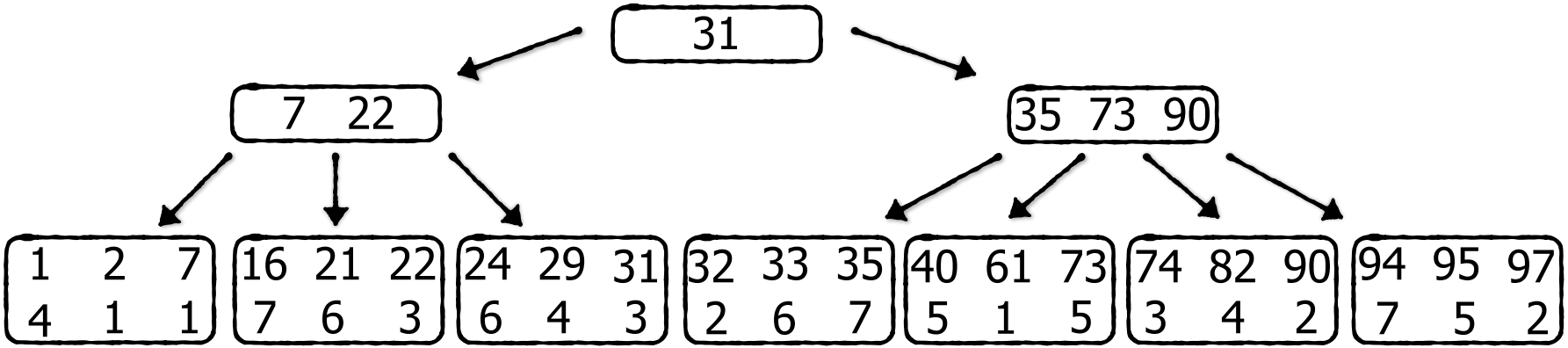

31
7 22
35 73 90
1 2 7
4 1 1
16 21 22
7 6 3
24 29 31
6 4 3
32 33 35
2 6 7
40 61 73
5 1 5
74 82 90
3 4 2
94 95 97
7 5 2

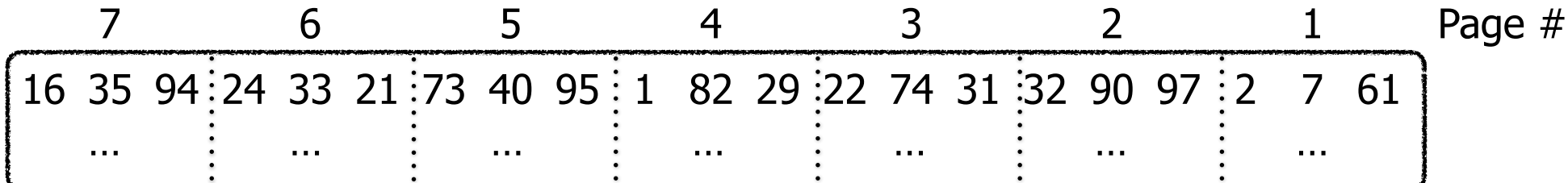

7
6
5
4
3
2
1
Page #
16 35 94
24 33 21
73 40 95
1 82 29
22 74 31
32 90 97
2 7 61
...
...
...
...
...
...
...

Search cost? **= depth of tree**

= $\log_B (N / B)$

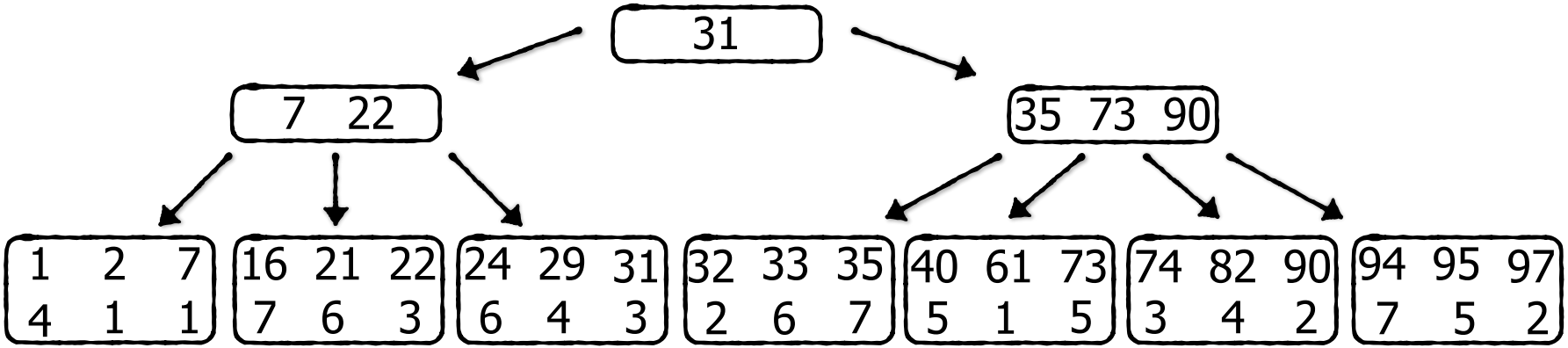
 # fanout **# leaves**

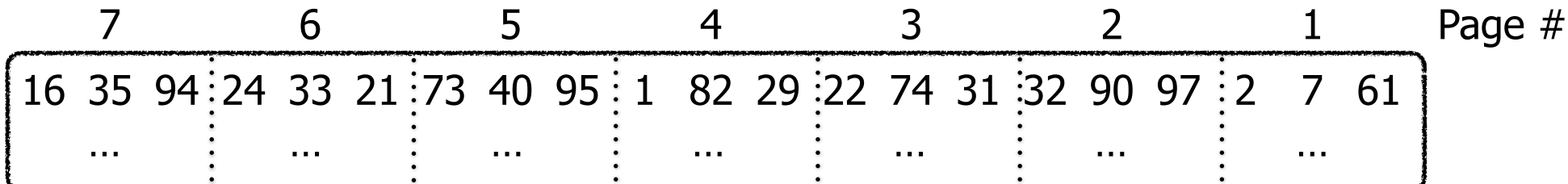




Search cost? **= depth of tree**

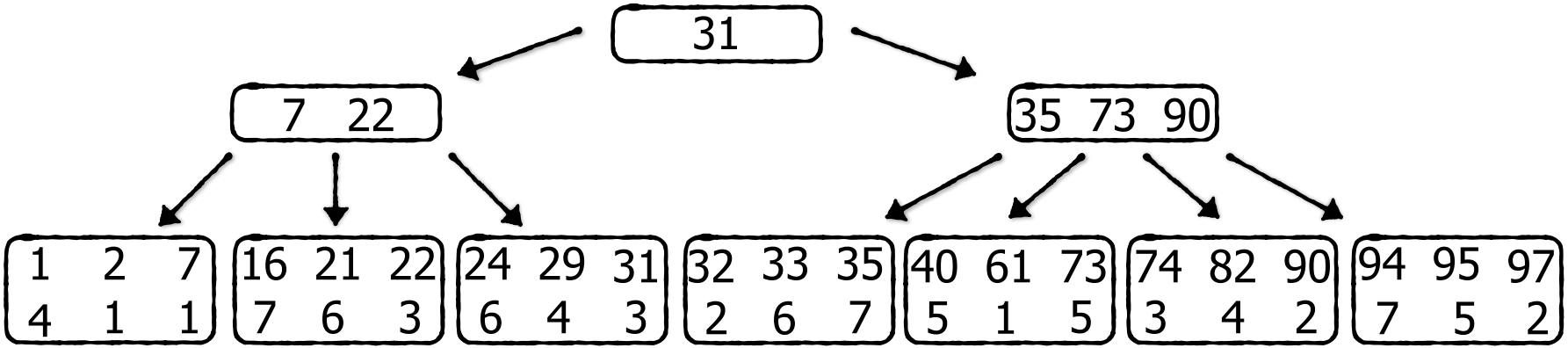
= $O(\log_B N)$ I/O

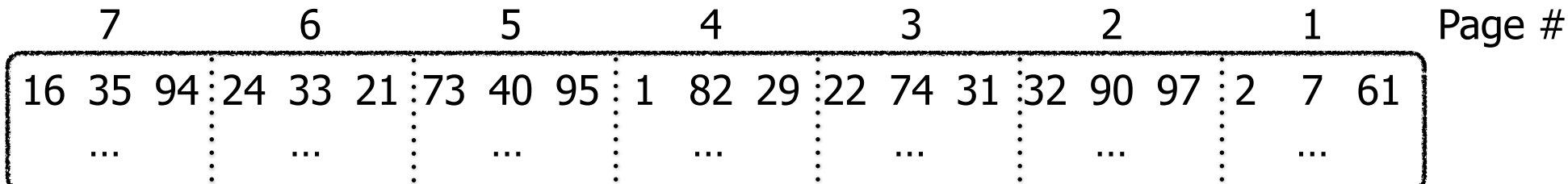




Search cost? $O(\log_B N)$ I/O

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

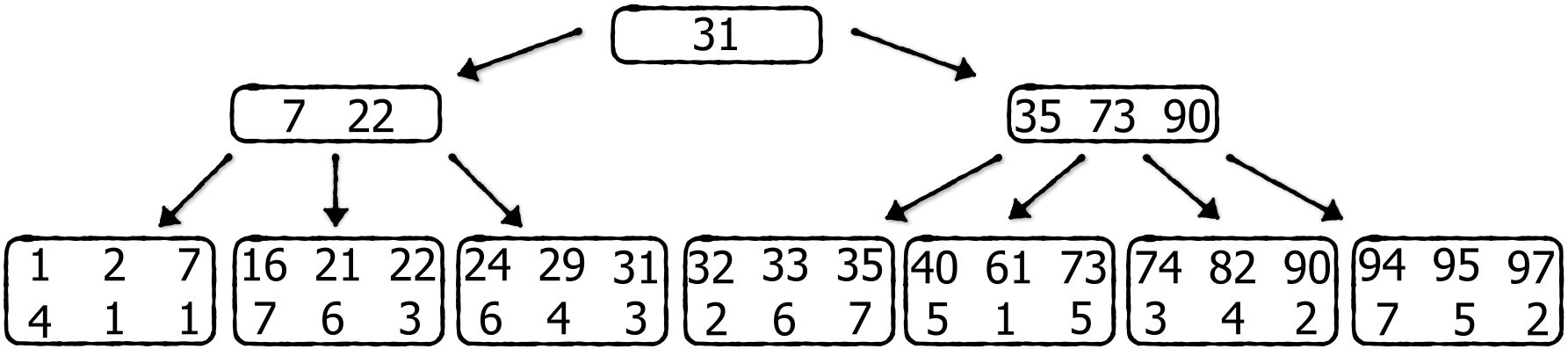


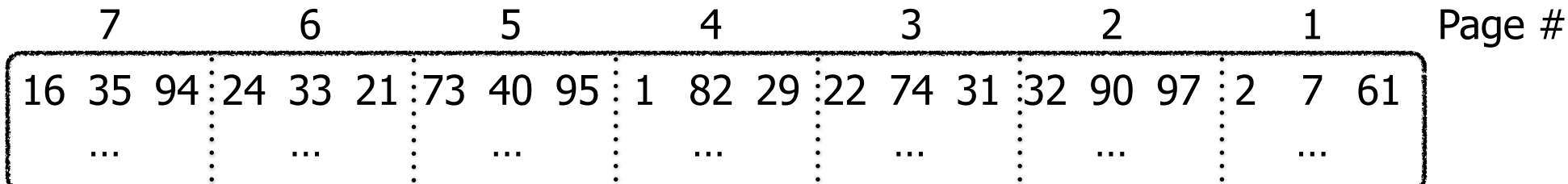


Search cost: $O(\log_B N)$ I/O

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?

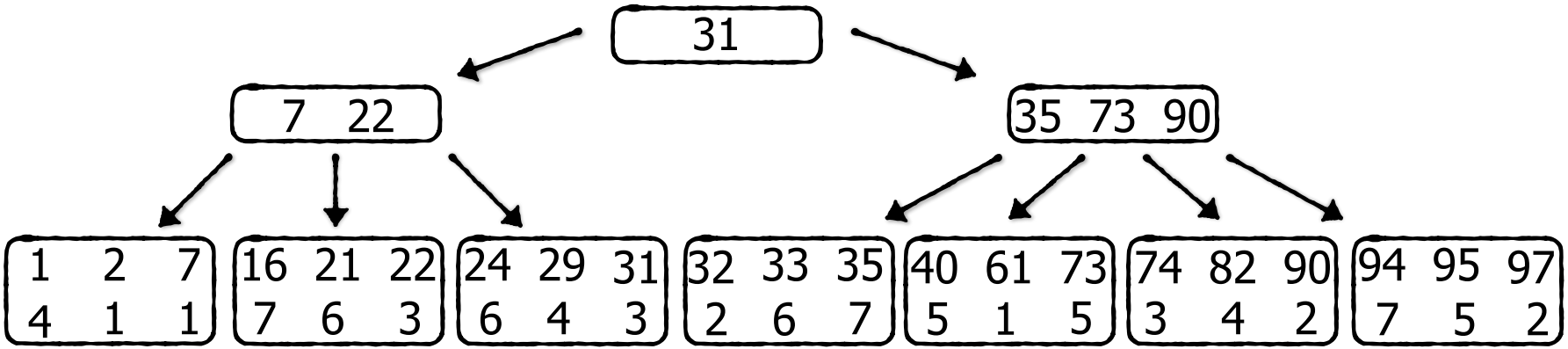


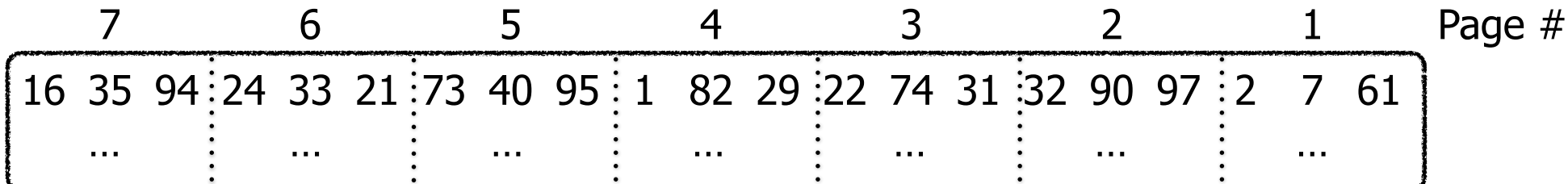


Search cost: $O(\log_B N)$ I/O

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? **$\log_B N = 3$**



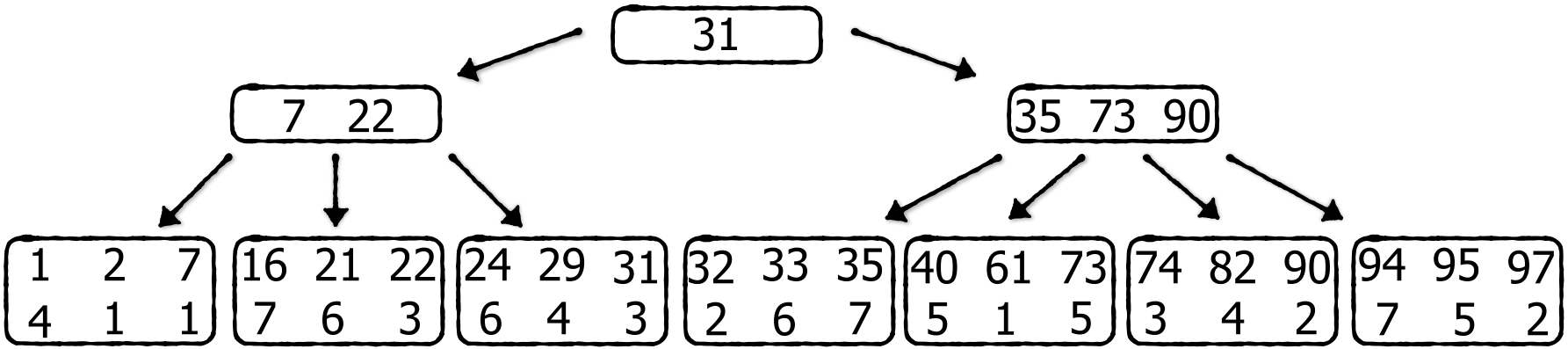


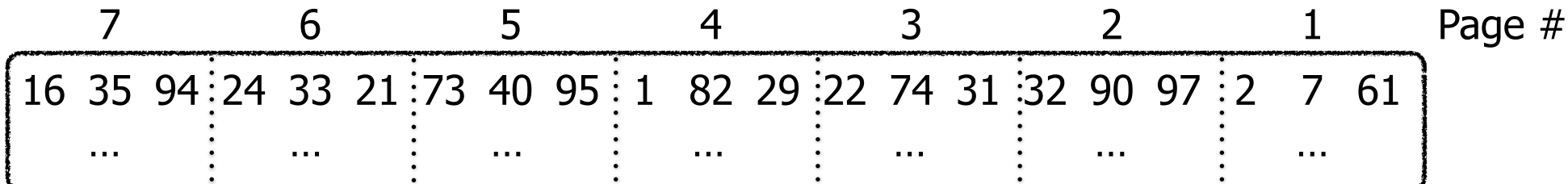
Search cost: $O(\log_B N)$ I/O

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? $\log_B N = 3$

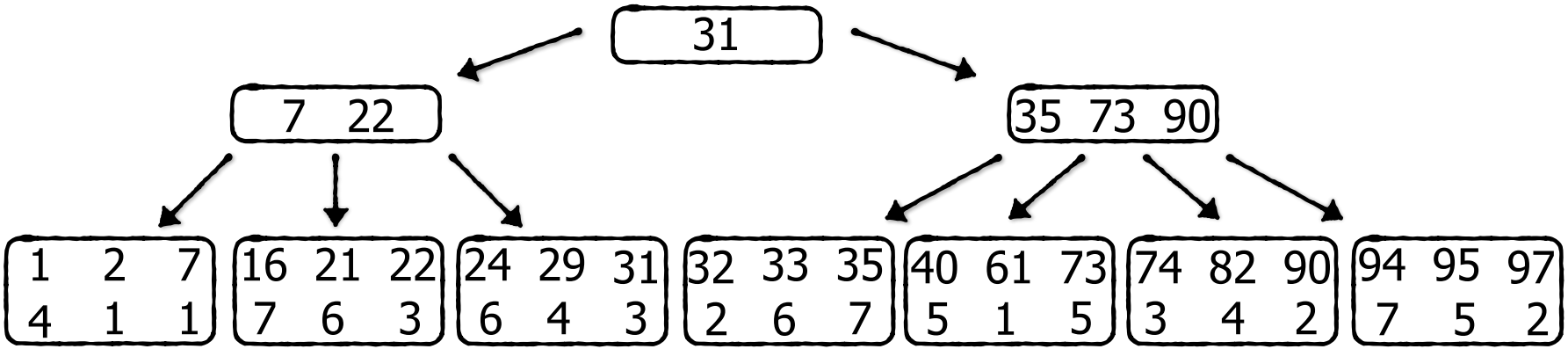
With a binary tree, it would have been far larger.

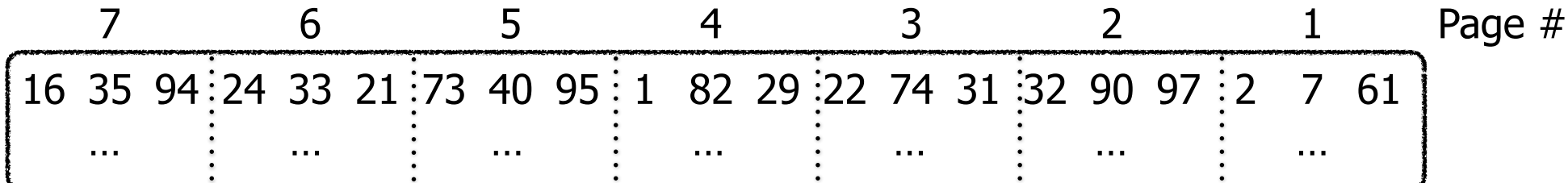




Search cost: $O(\log_B N)$ I/O

How many CPU comparisons?

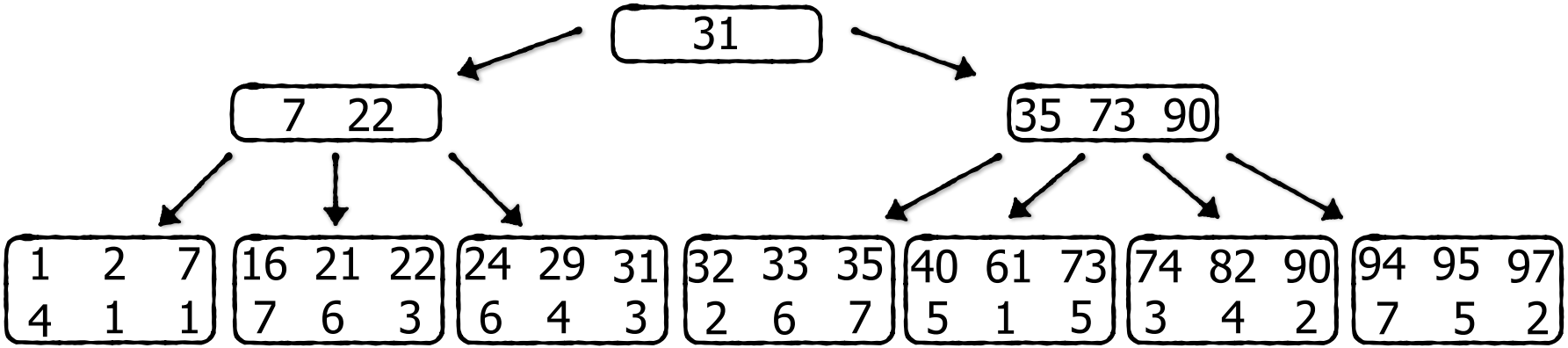


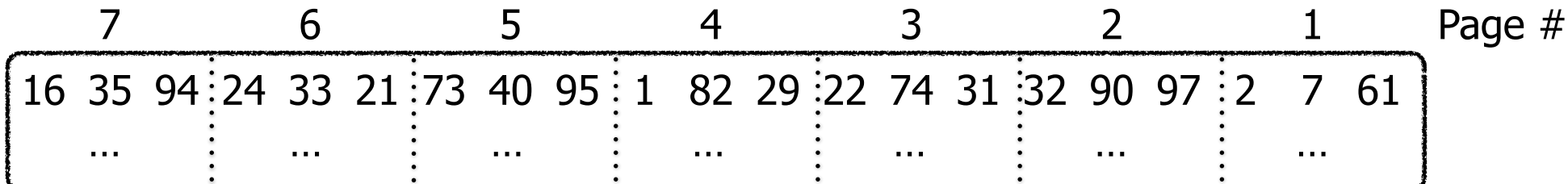


Search cost: $O(\log_B N)$ I/O

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

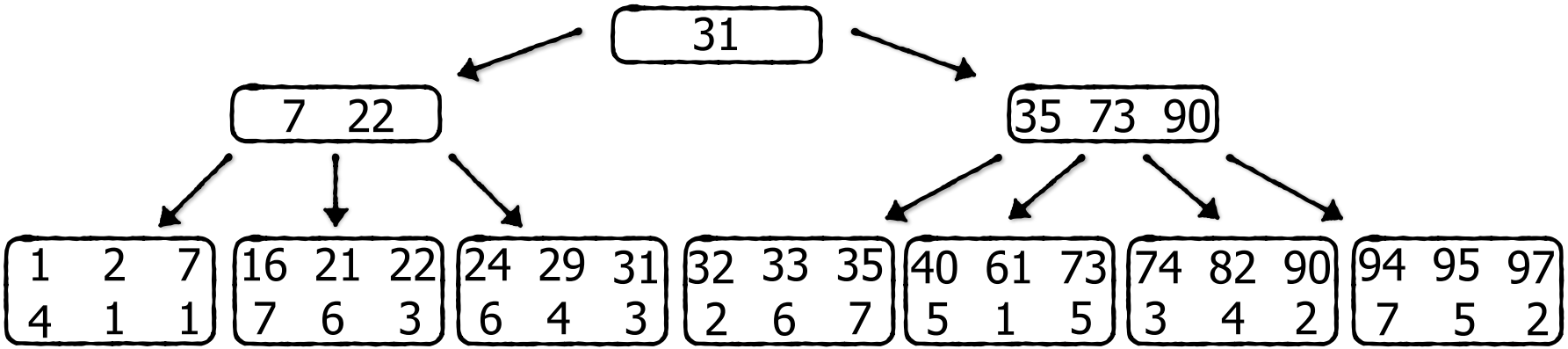
$\log_B N$: # pages searched; $\log_2 B$: binary search

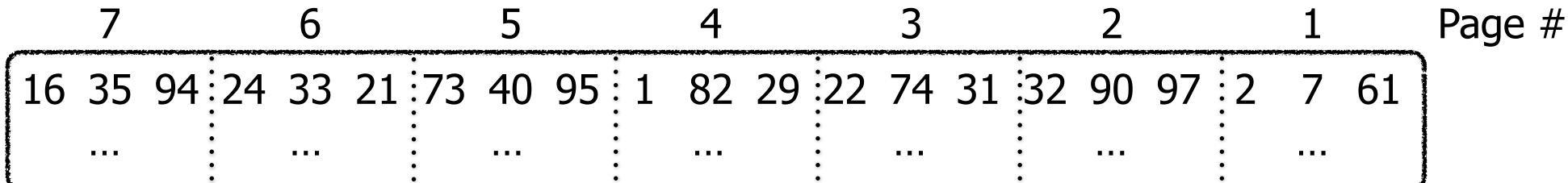




Search cost: $O(\log_B N)$ I/O

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

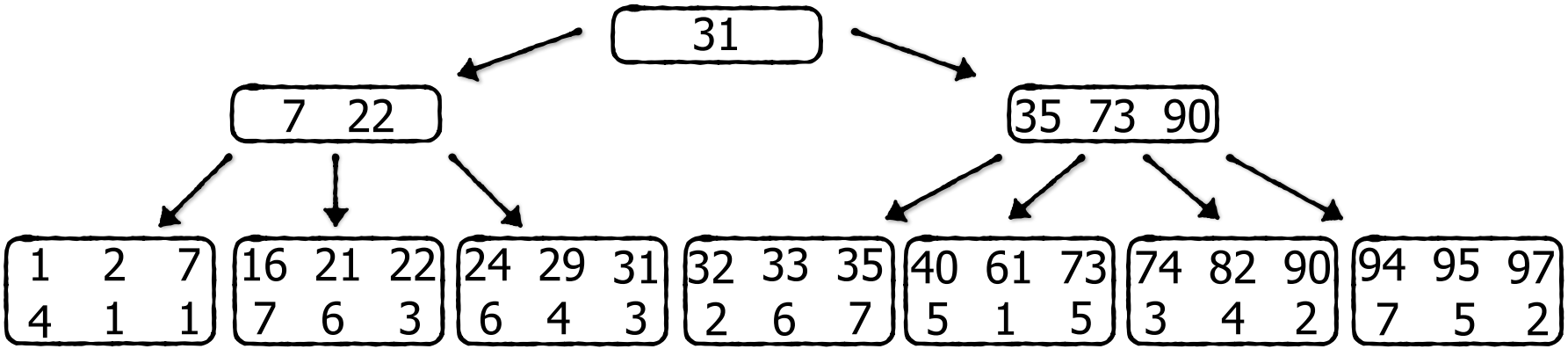


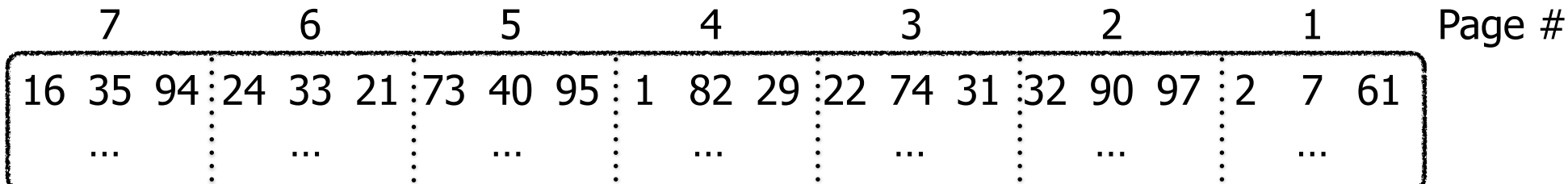


Both are a part of search cost

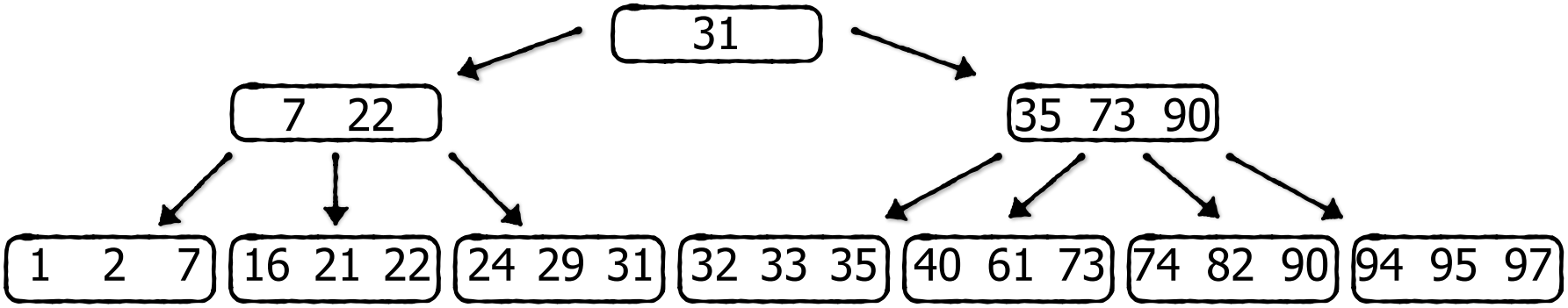
Search cost: $O(\log_B N)$ I/O ← But this is the bottleneck

How many CPU comparisons? $O(\log_2 N)$ ← And this usually isn't



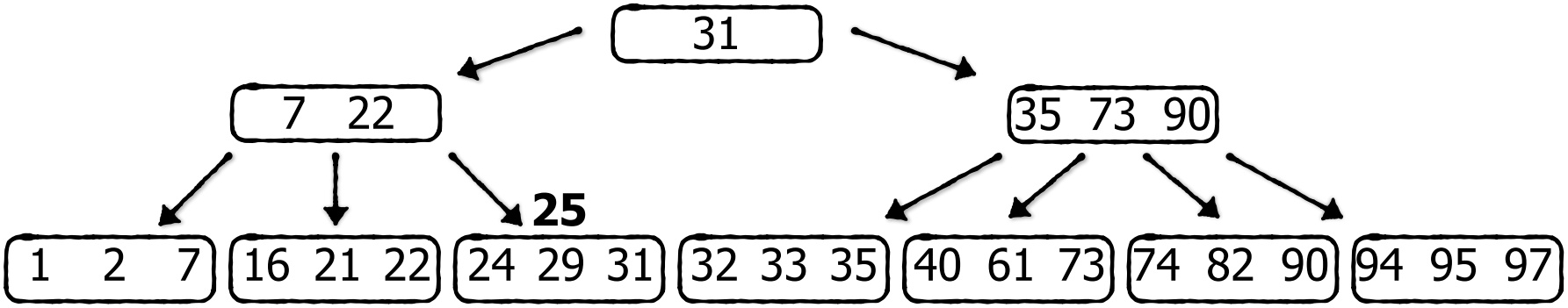


How to insert? **e.g., key 25**



How to insert?

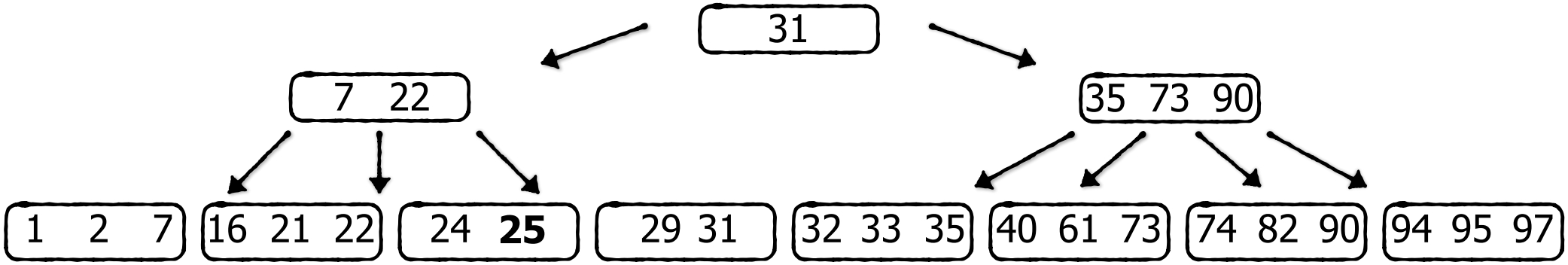
First find target node



How to insert?

First find target node

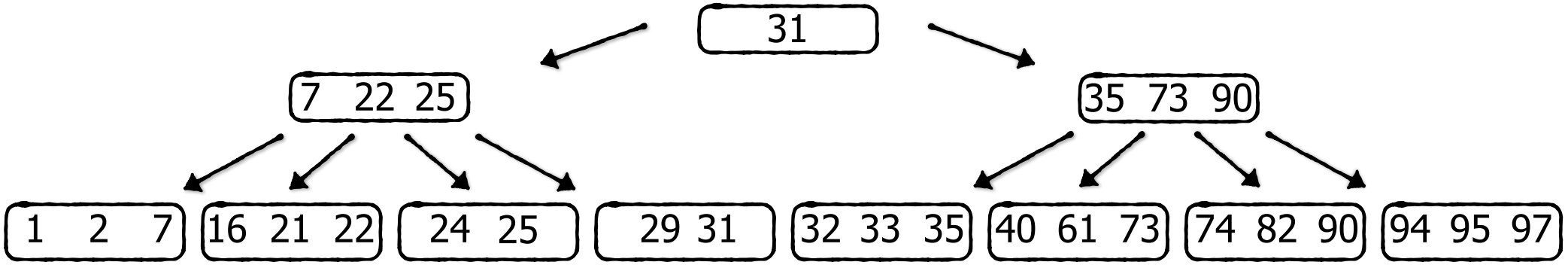
If node is full, split target node



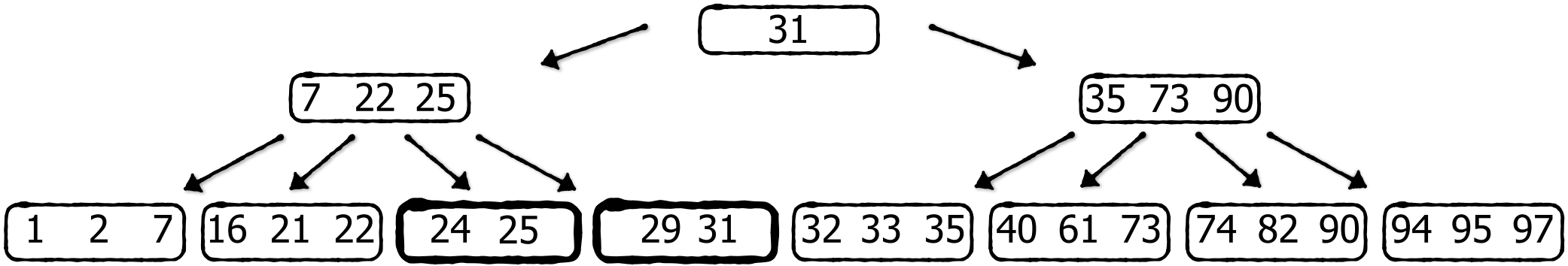
How to insert?

First find target node

Connect new node to parent



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



read I/Os per insertion? $O(\log_B N)$

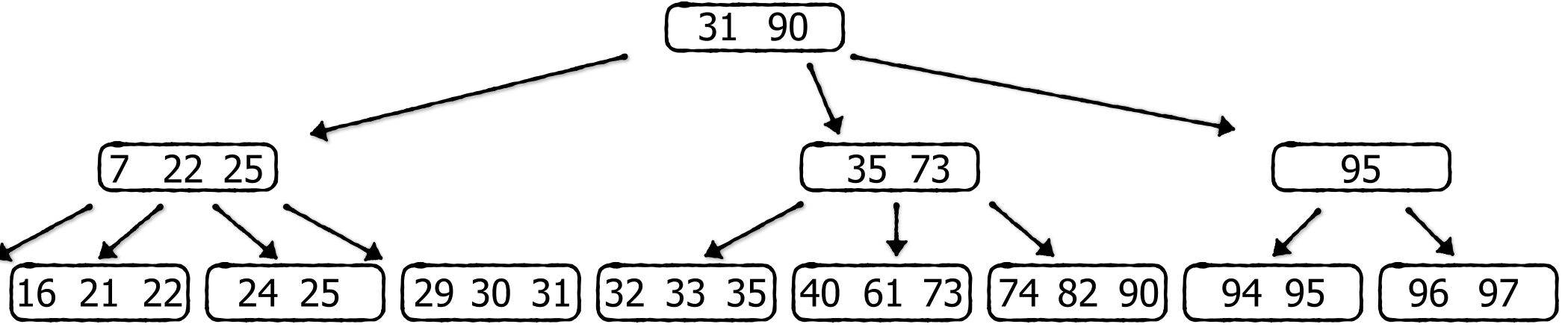
 # write I/Os per insertion?

Intuition: **Every insertion updates 1 leaf**

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

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

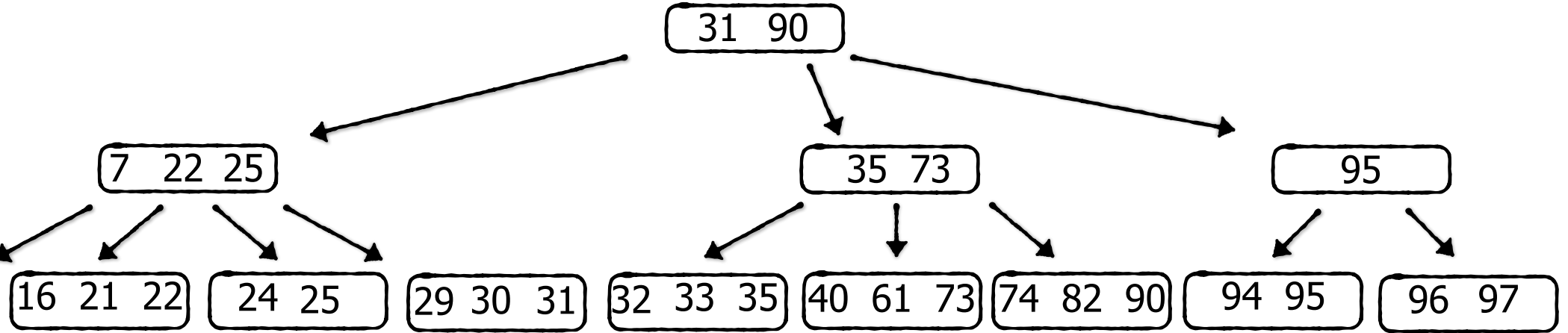
one in $O(B^{-3})$ insertions triggers grandparent split



How many read I/Os per insertion? $O(\log_B N)$

How many write I/Os per insertion?

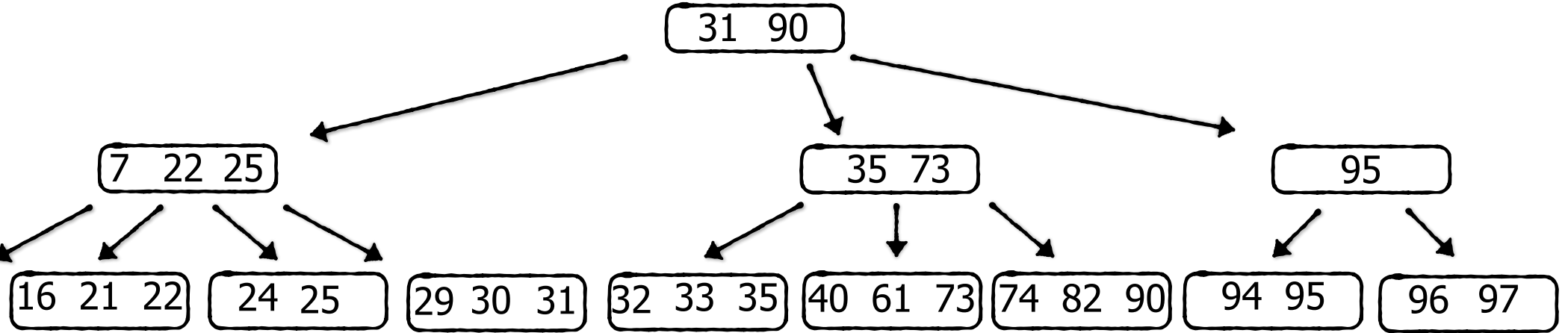
$$\mathbf{1 + B^{-1} + B^{-2} + B^{-3} + \ldots = O(1)}$$



An insertion costs	$O(\log_B N)$	Read I/Os
	O(1)	Write I/O

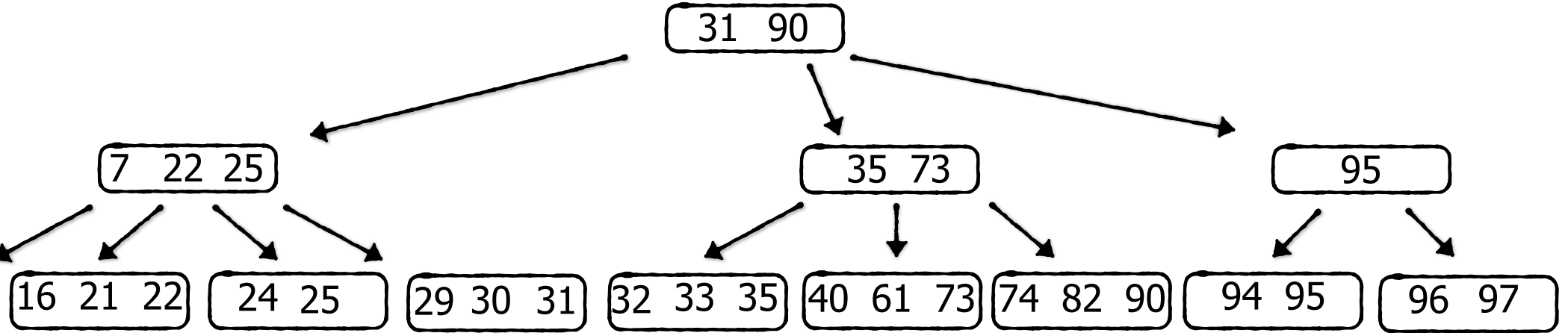
On disk, read/write costs are symmetric.

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



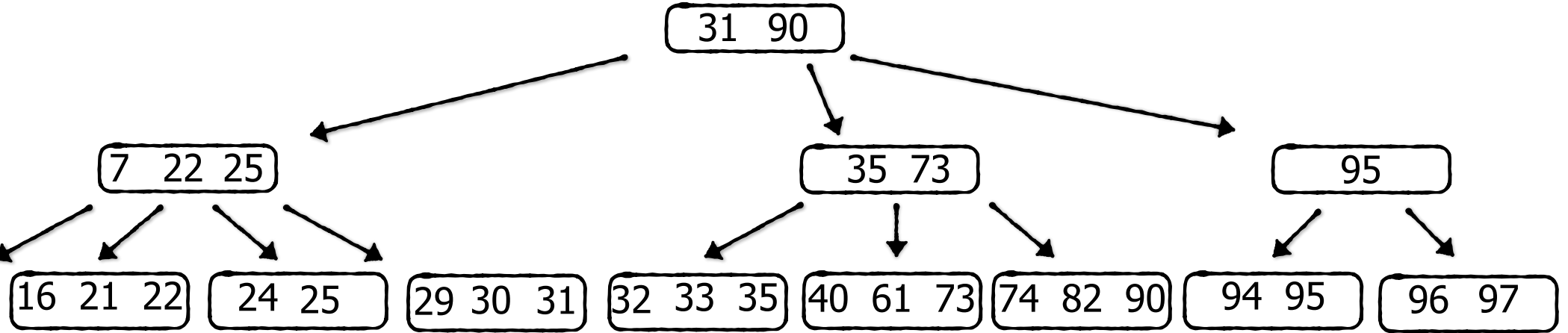
An insertion costs	$O(\log_B N)$	Read I/Os
	O(1)	Write I/O

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

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



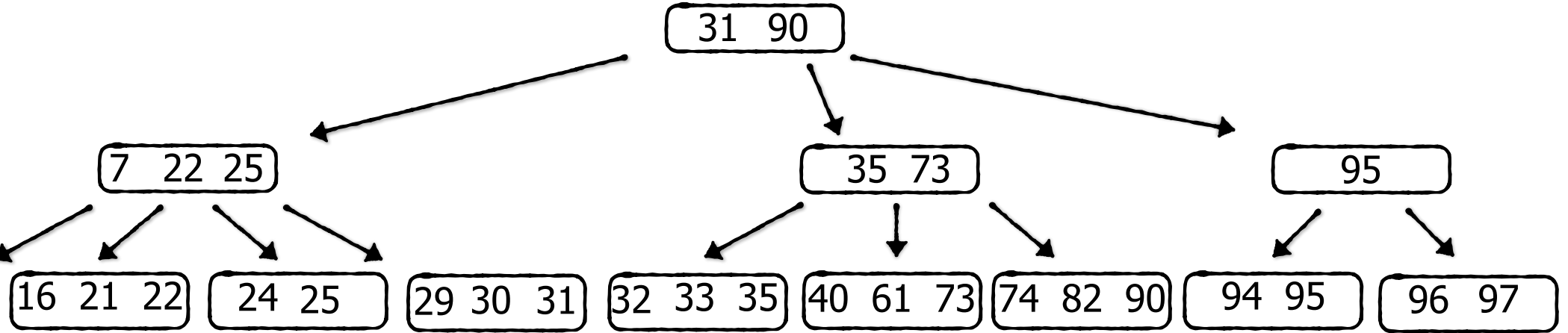
Indexes speed up queries. Do they have downsides?

(1) They take up storage space

(2) They must be updated as data changes

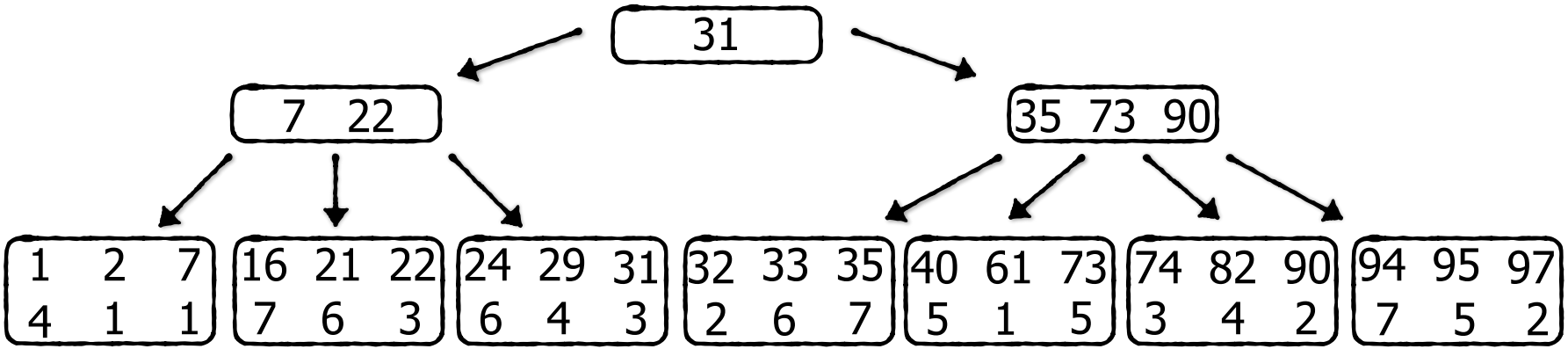
To index or not to index? That depends on the workload.

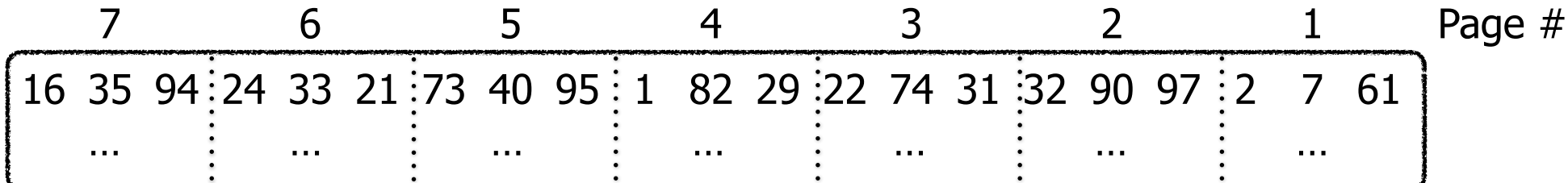
More on Thursday



How do range scans work?

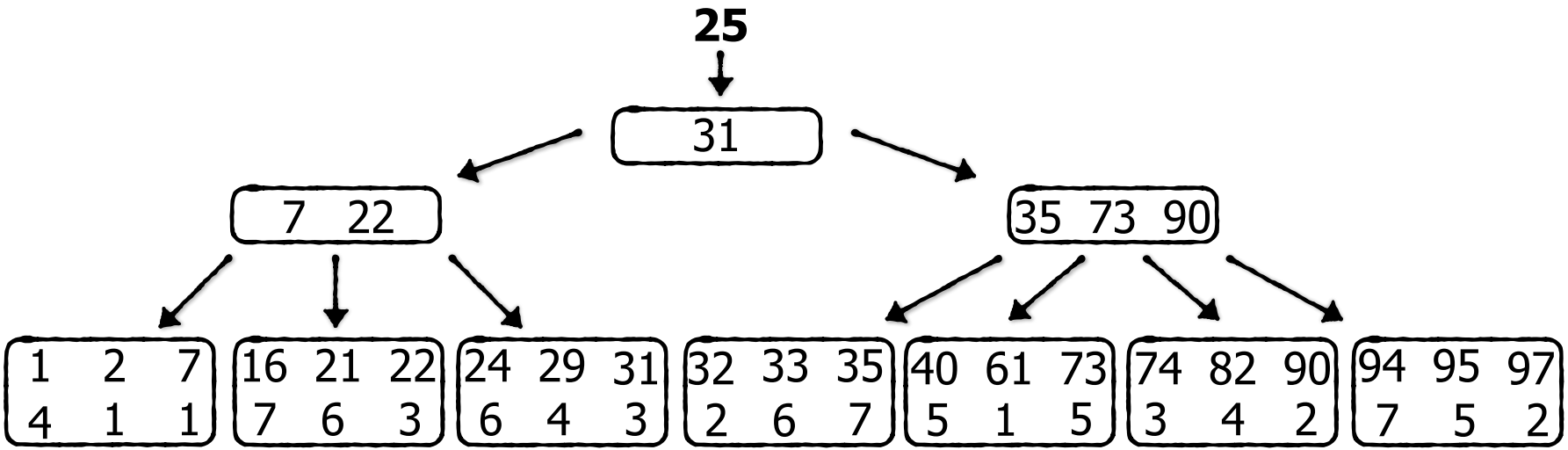
Select * from table where A > 25 and A < 35

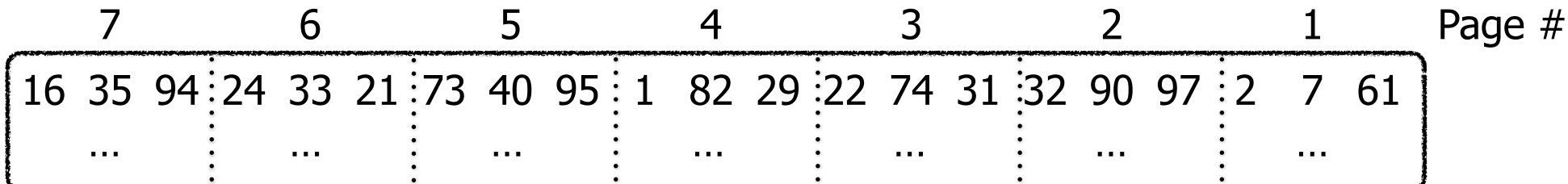




How do range scans work?

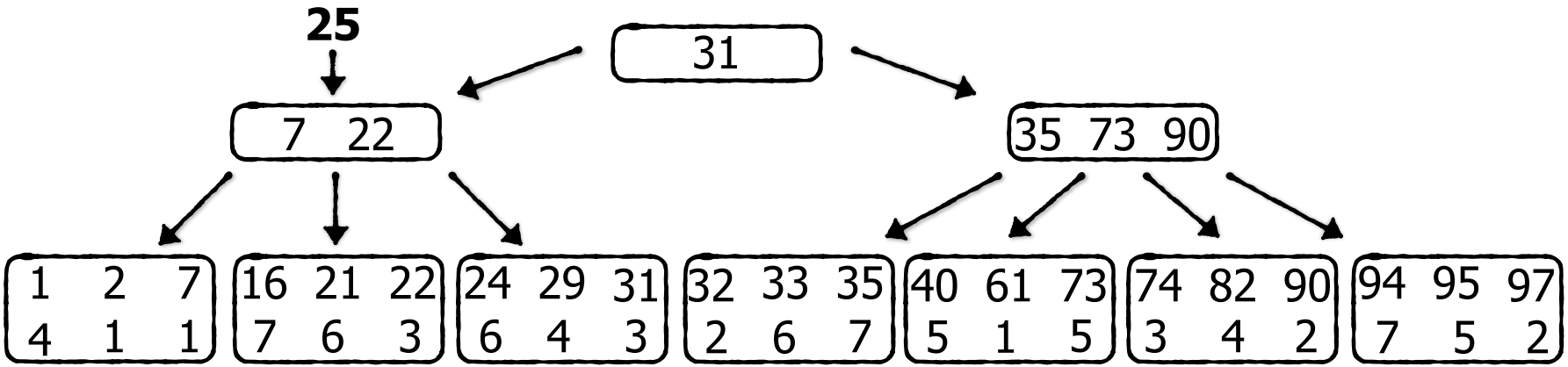
Select * from table where A > 25 and A < 35

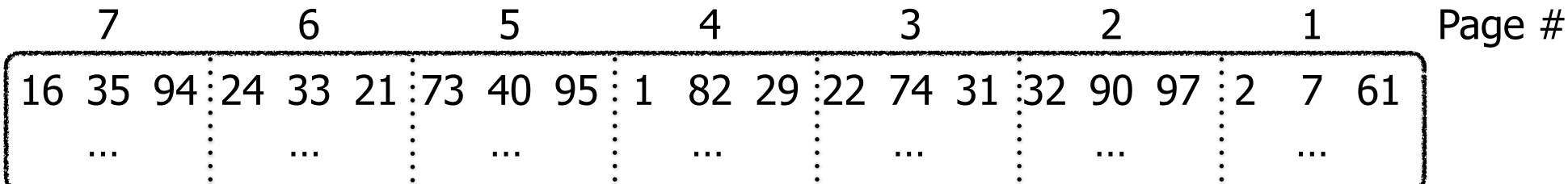




How do range scans work?

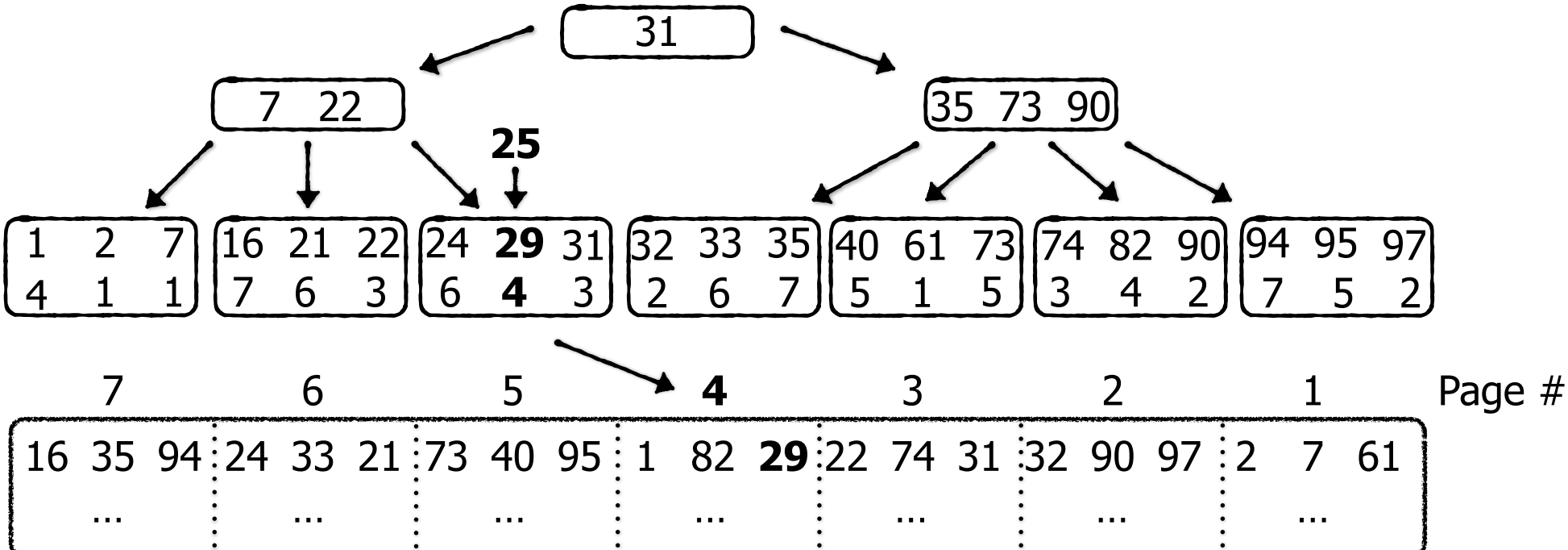
Select * from table where A > 25 and A < 35





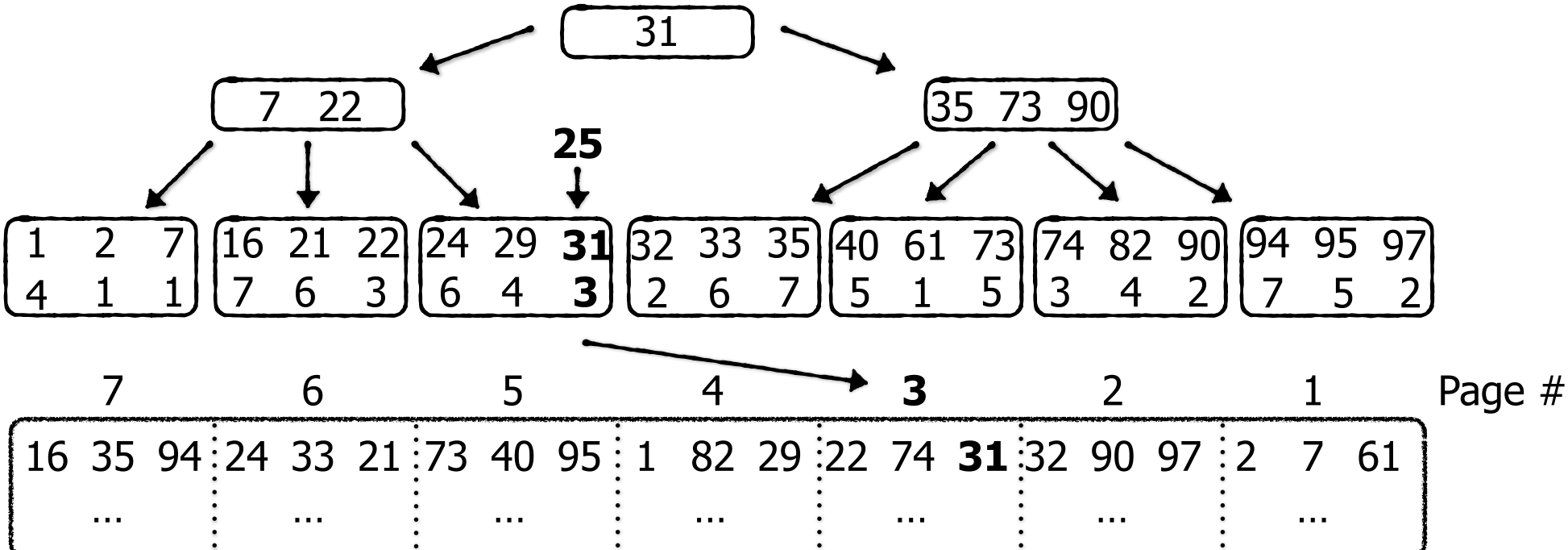
How do range scans work?

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



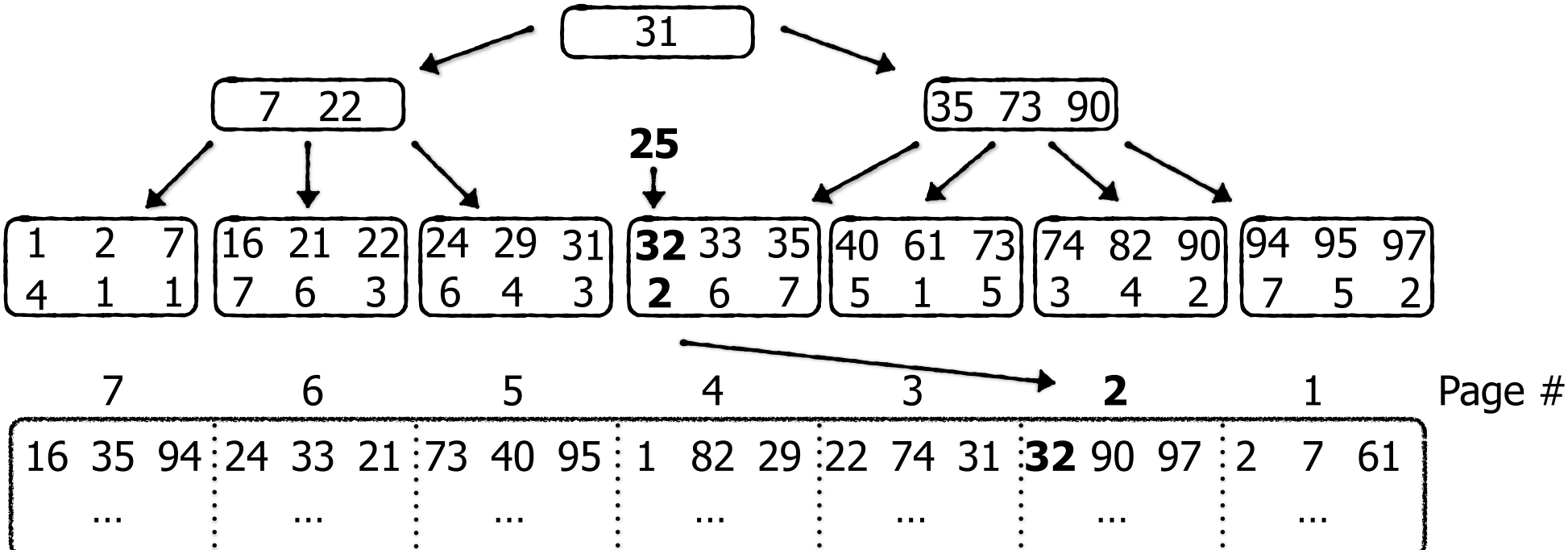
How do range scans work?

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



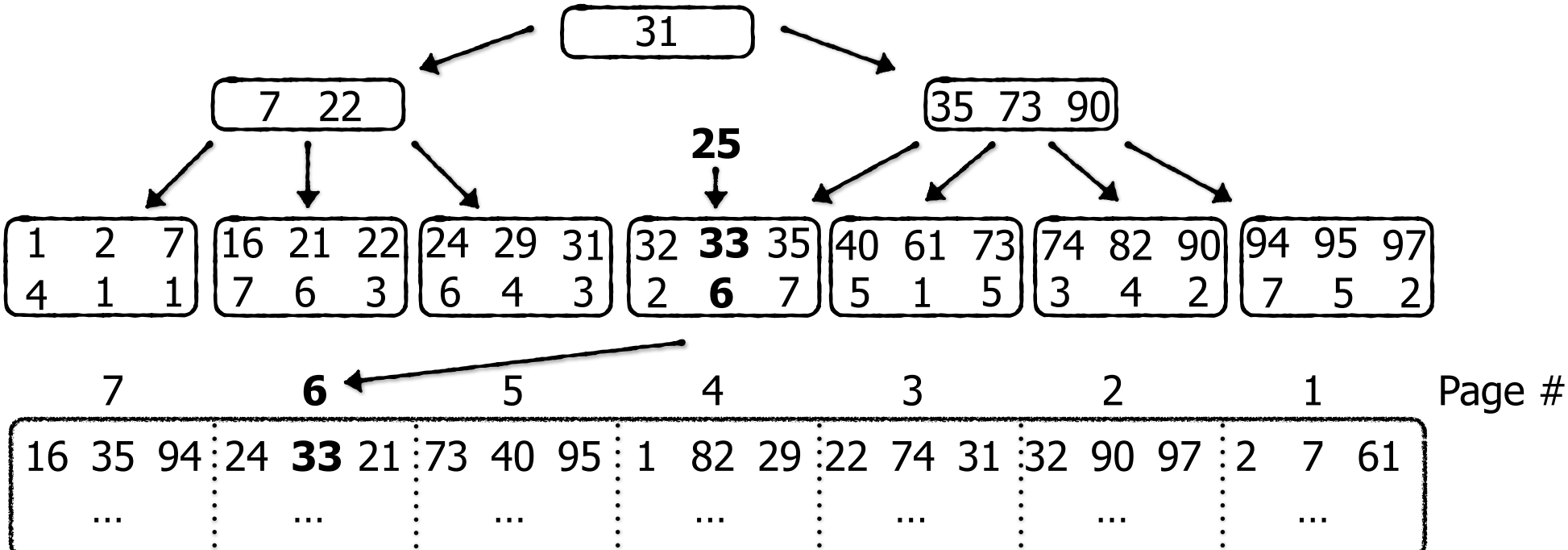
How do range scans work?

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

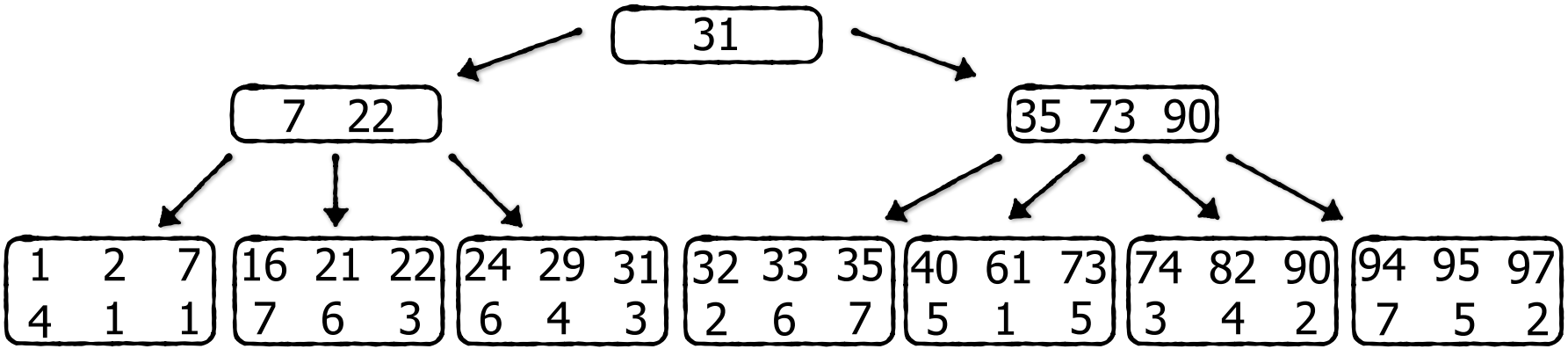


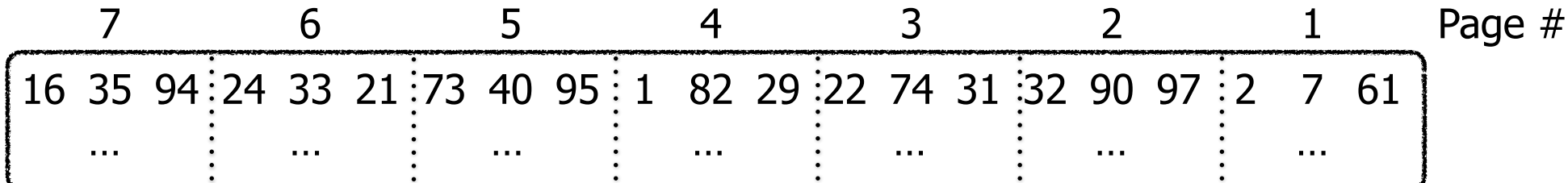
How do range scans work?

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



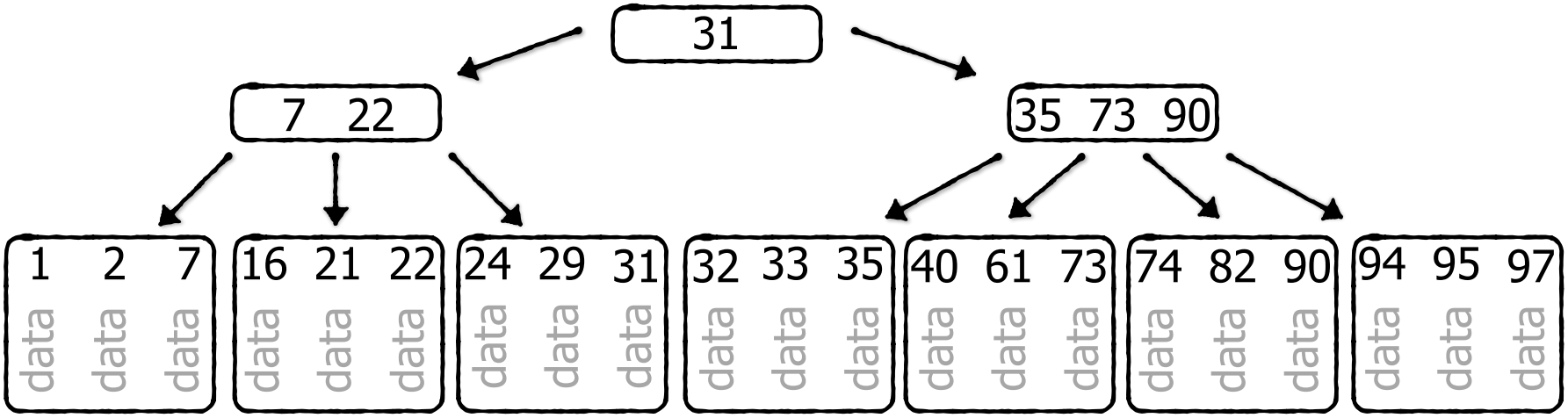
Scan Cost? **O($\log_B$ N + S) I/Os** **S = # entries in the range**



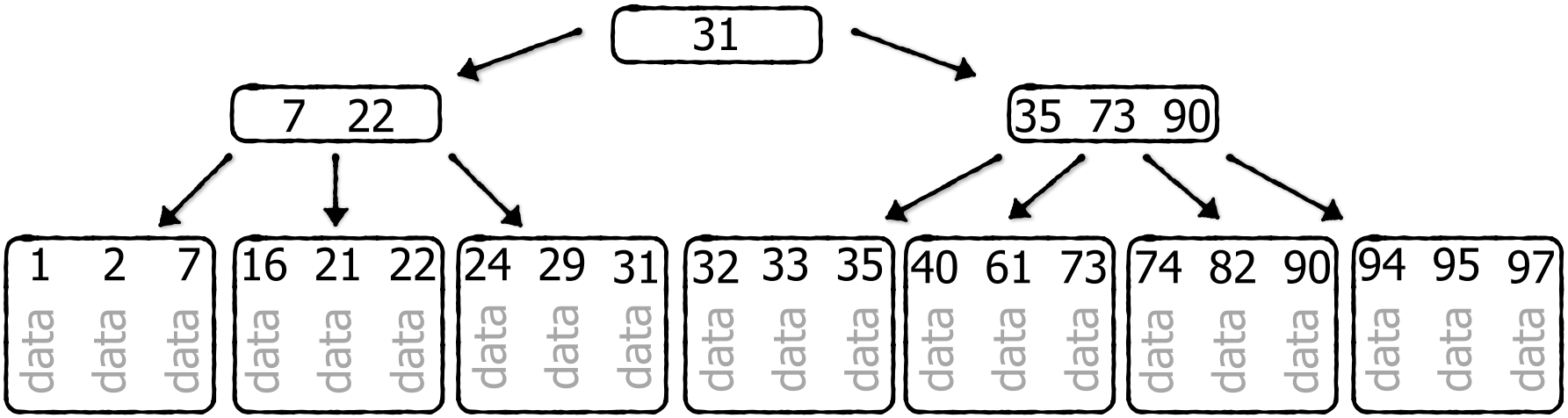


Scan Cost? $O(\log_B N + S)$ I/Os S = # entries in the range

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



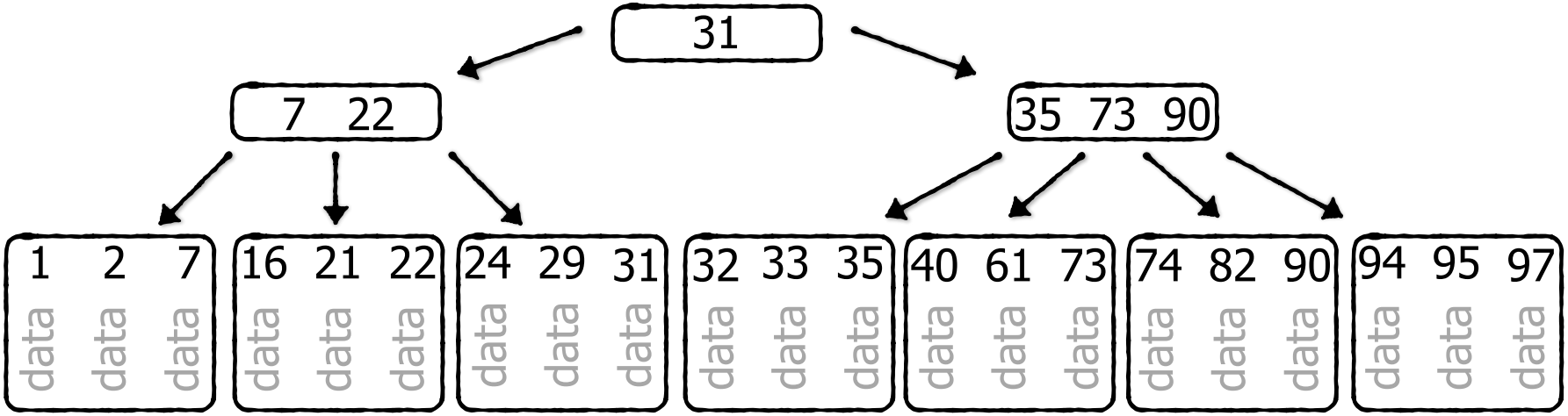
This is known as a "Clustered Index"



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

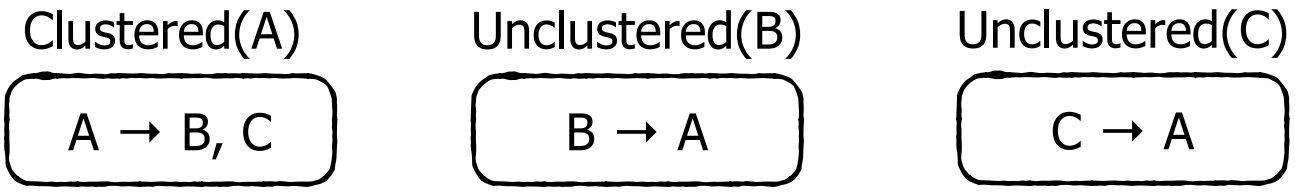
Clustered Index cost model? **$O(\log_B N + S/B)$**

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:



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-Tree & Filters