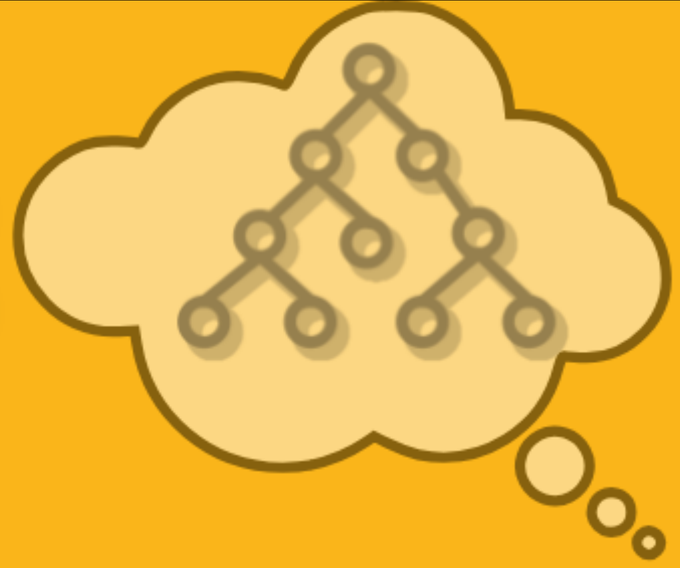
DATABASE INDEXES

B & B+ TREES



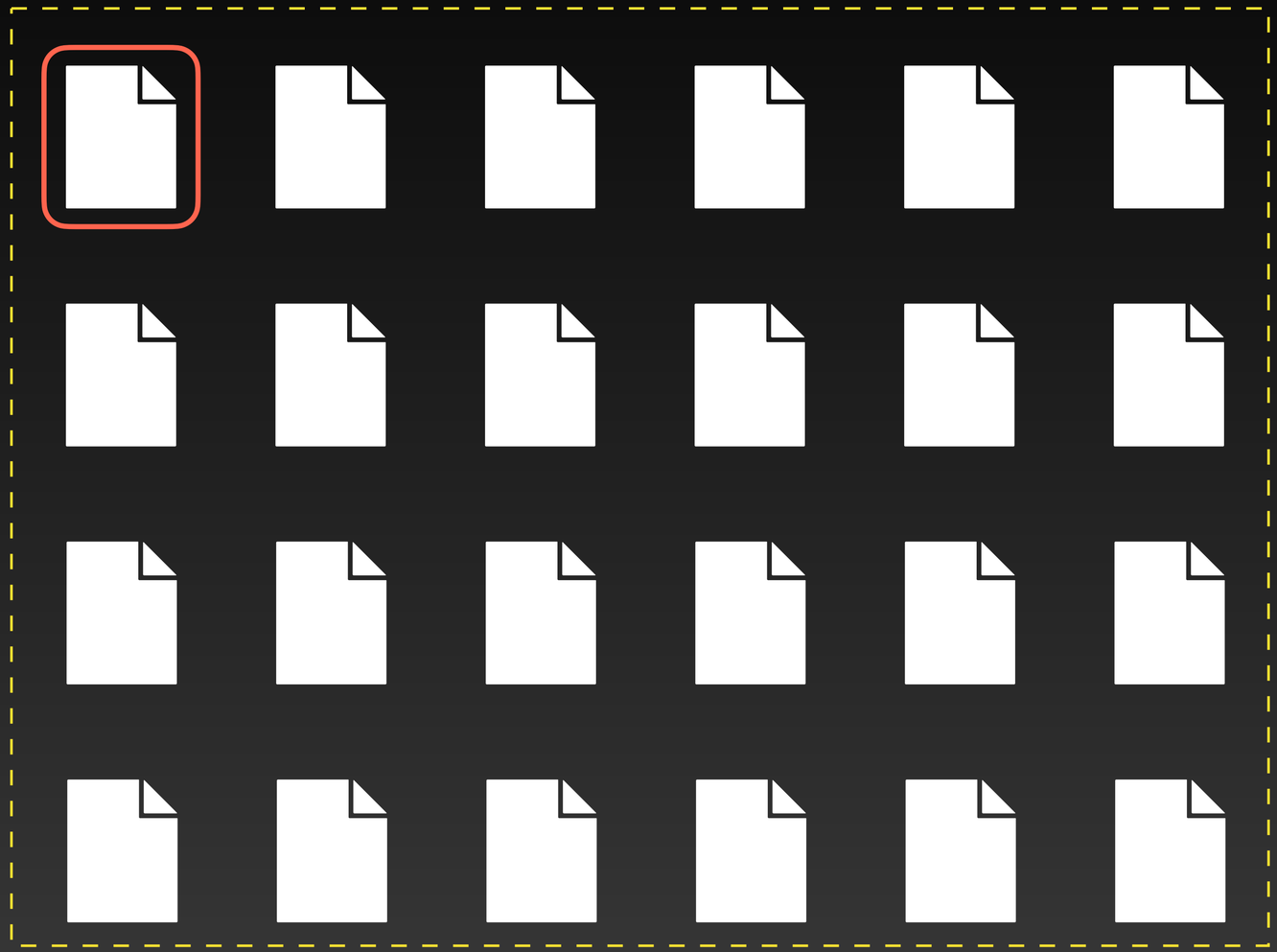
Amr Elhelw's TECH VAULT

the following are set of ways to find a row:

Find rows with x=25

All pages

for table T



normal scanning for all rows
in all pages related to a table

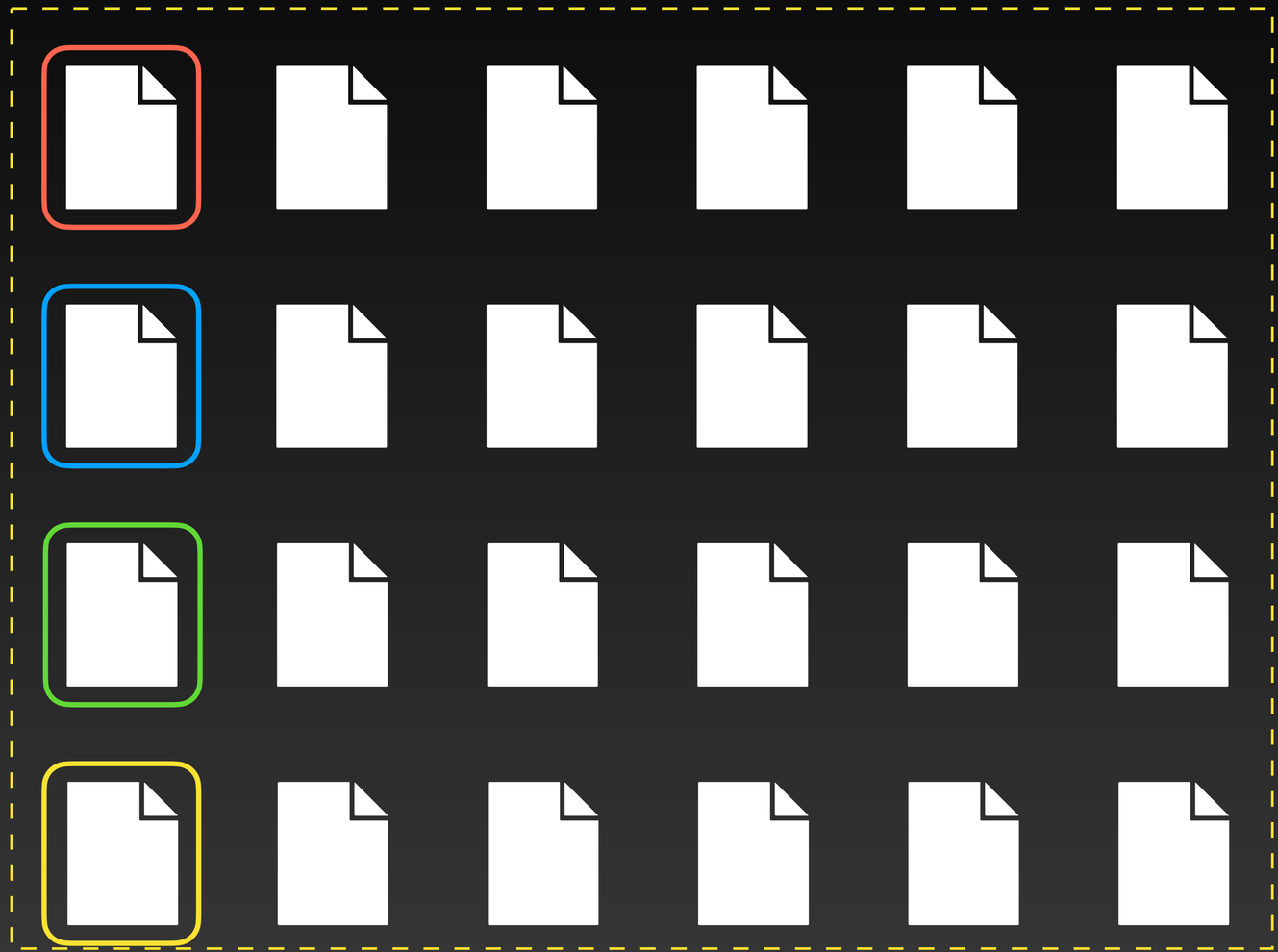
cons: too slow

Parallel Scanning

All pages

for table T

(multiple threads)



cons:

---

* complex in handling threads
* may cause bottle nicks
* still read all pages

Partitioning

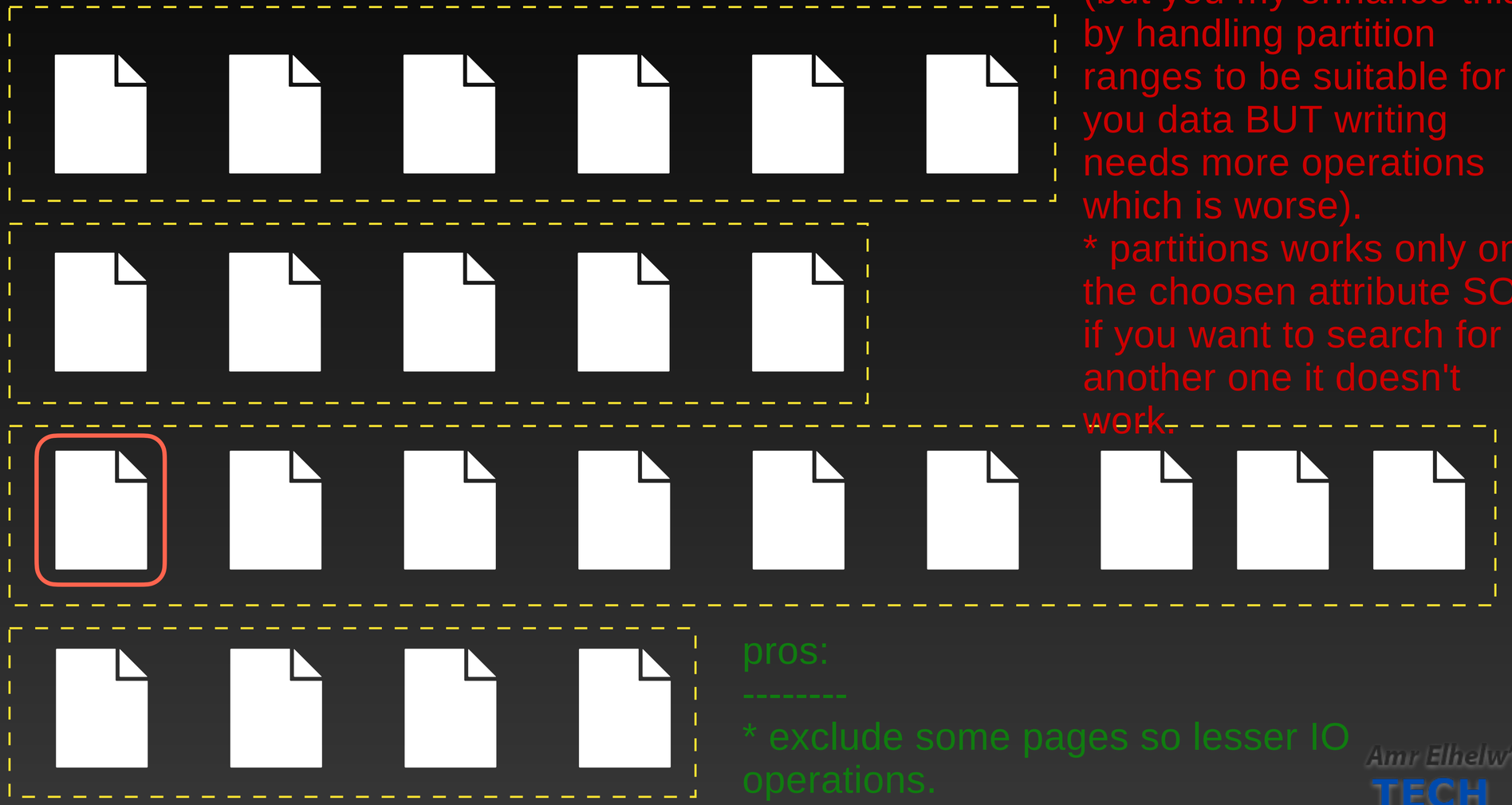
x = 25 ?

0 ≤ x < 10

10 ≤ x < 20

20 ≤ x < 30

30 ≤ x < 40



divide table into multiple partitions/tables according to the value of x

pros:

---

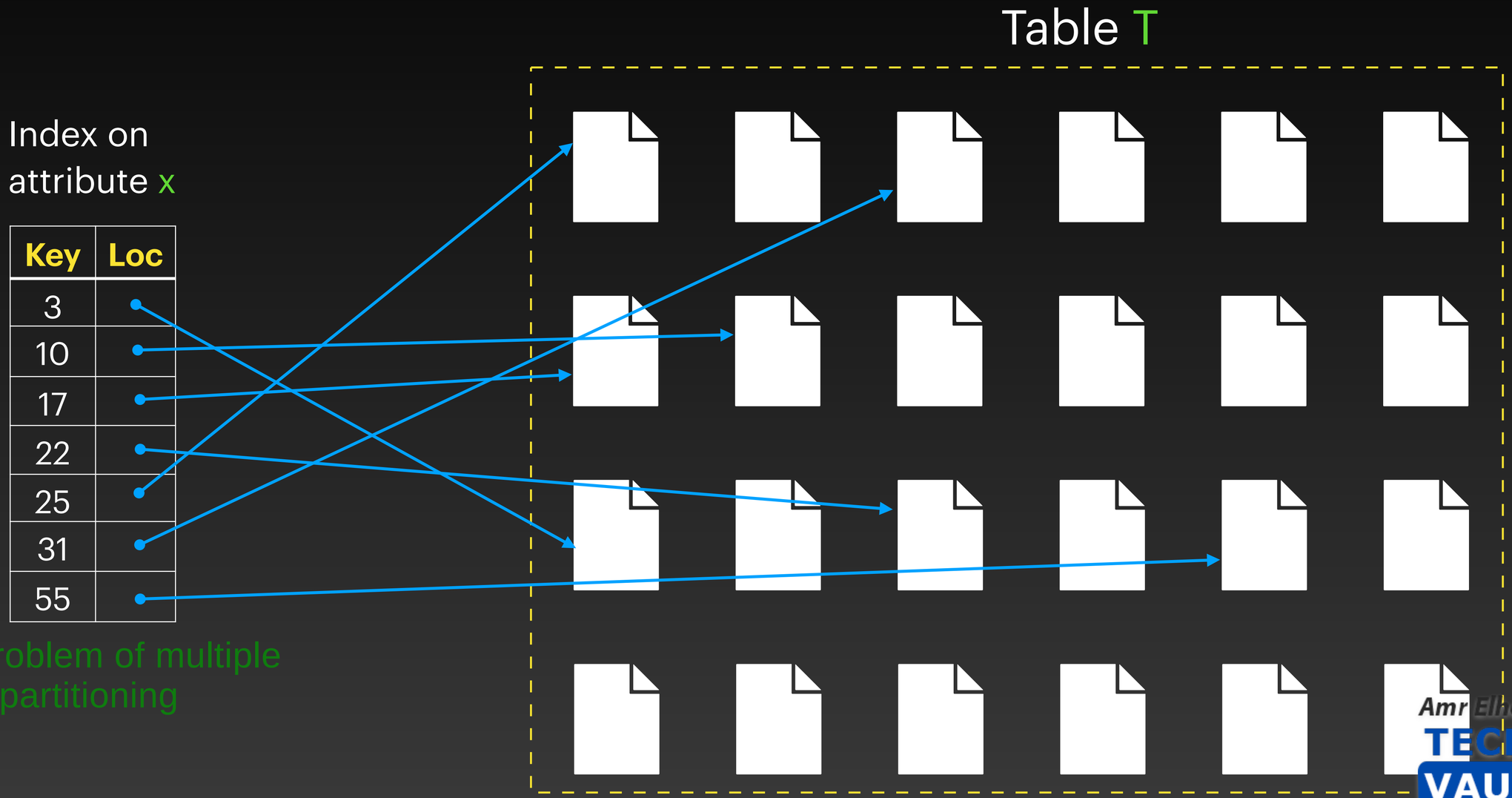
* exclude some pages so lesser IO operations.

Cons:

* not all partitions have the same count of rows because it depends on data and value of x. (but you my enhance this by handling partition ranges to be suitable for you data BUT writing needs more operations which is worse).
* partitions works only on the choosen attribute SO if you want to search for another one it doesn't work.

Amr Elhelw's TECH VAULT

Index



solves the problem of multiple attributes of partitioning fits queries:
* equality
* ranges

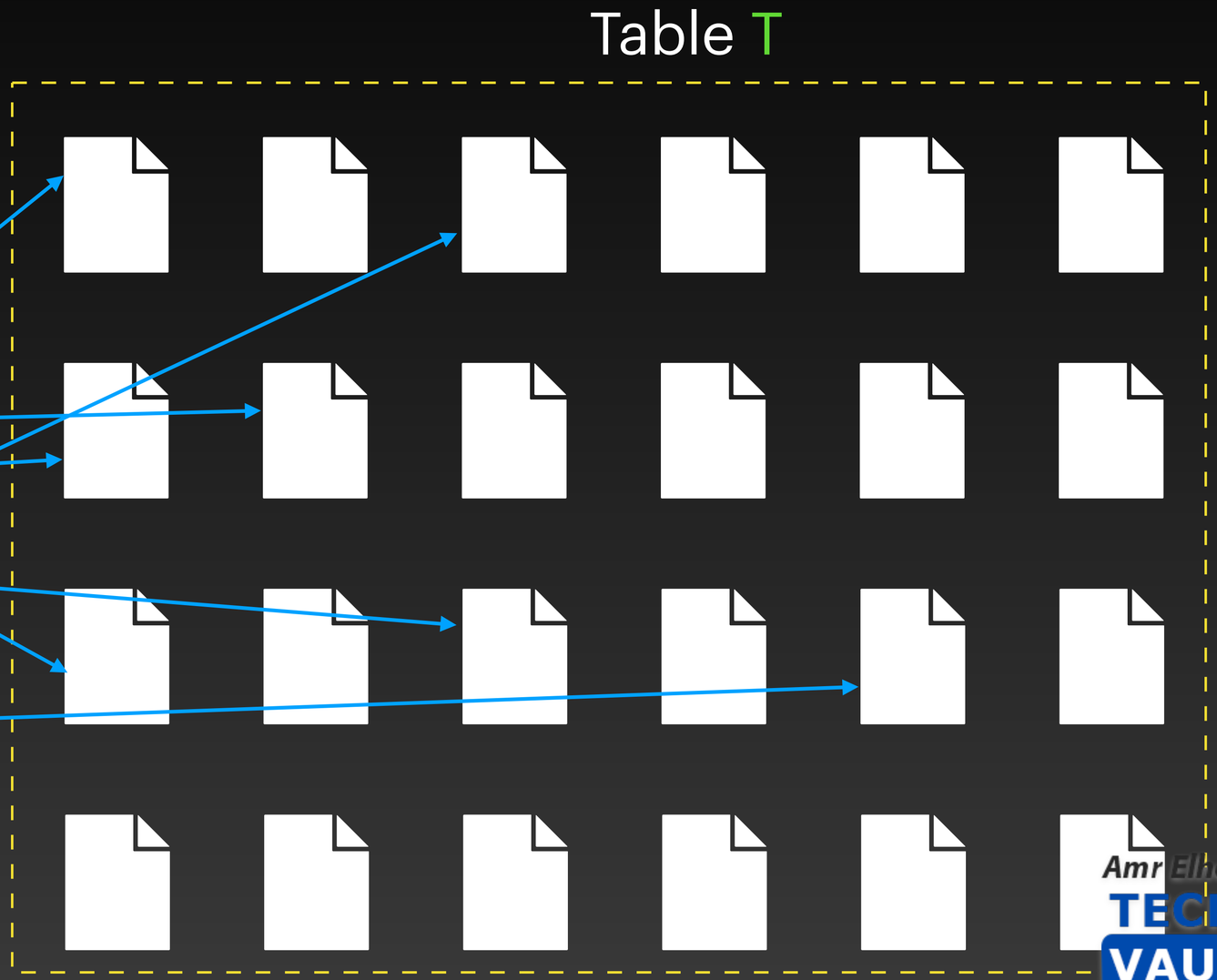
cons: indexes are also stored in pages hat need to be loaded to get your keys.
binary search solution may cause random access for pages with is expensive on disk.
insertion needs to find the pages then shifting all the others O(n)

Naive Approach - List

x = 25 ?

Index on attribute x

Key	Loc
3	
10	
17	
22	
25	
31	
55	

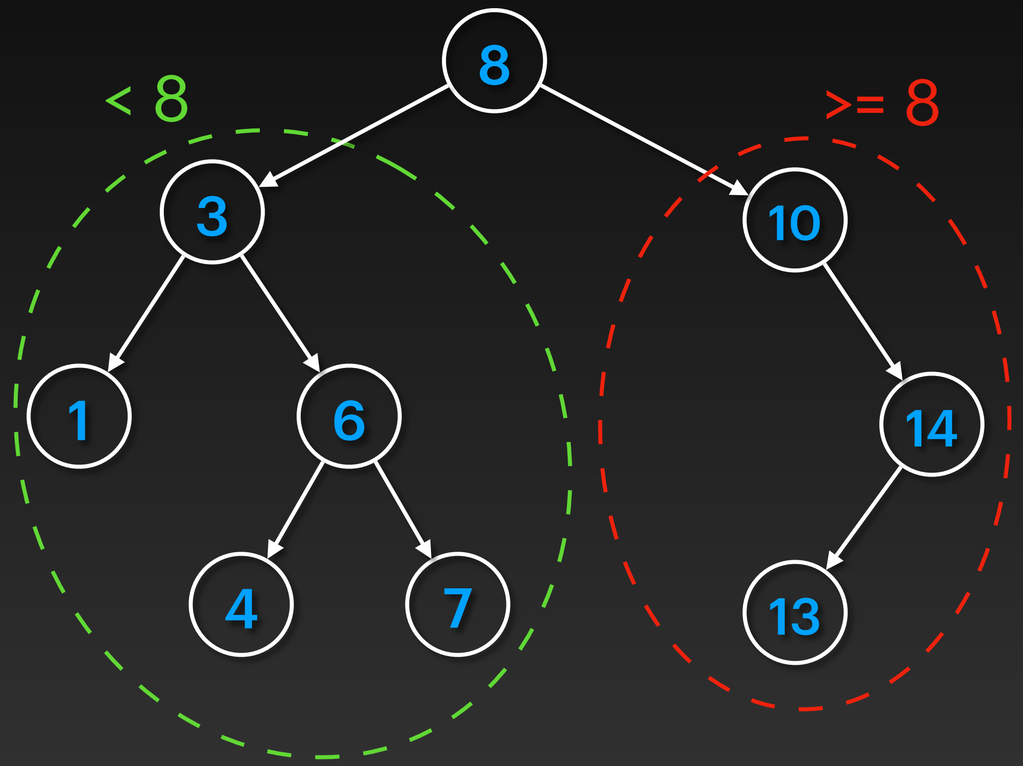


Search: O(n)

Binary Search Tree (BST)

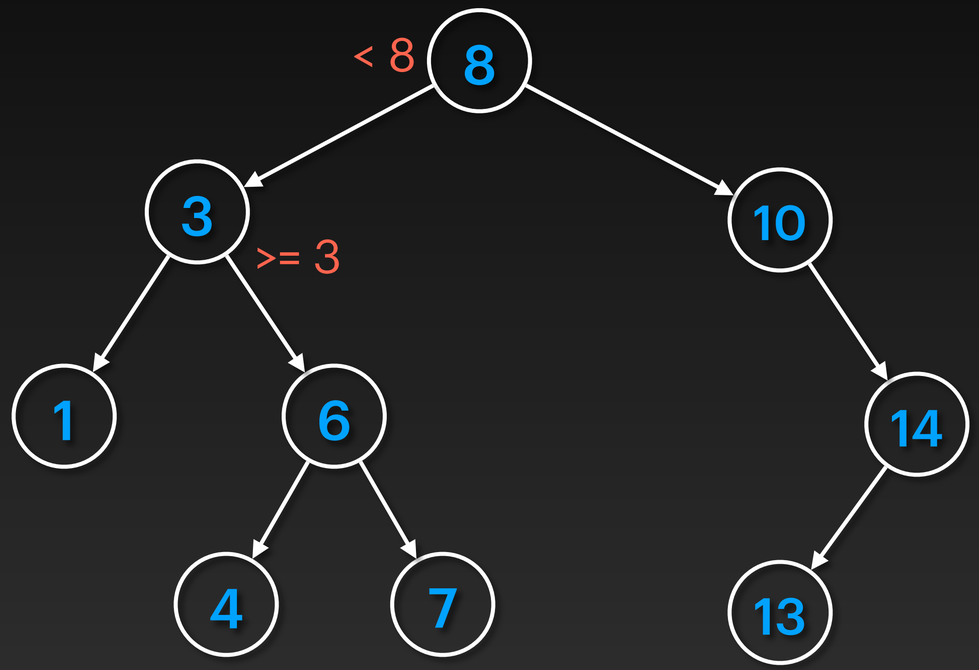
solution for the O(n) search

puts keys as nodes

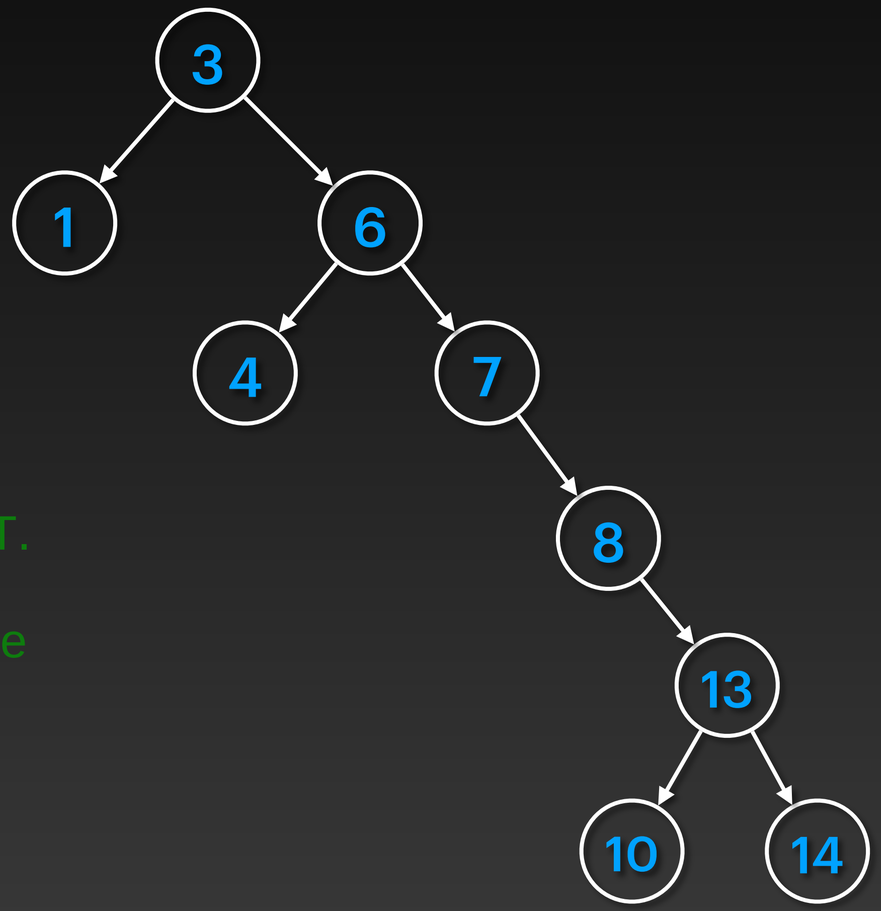


Binary Search Tree (BST)

x = 6 ?



Binary Search Tree (BST)



Cons:
* in worst case it may cause linear access if the tree is not balanced.

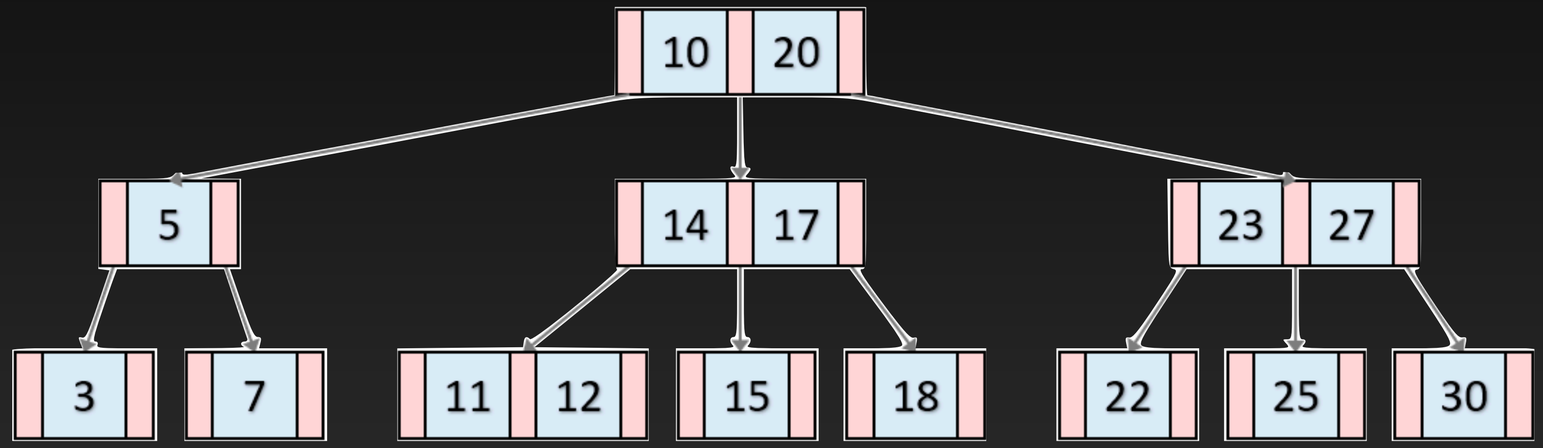
- as solution use BBST instead of BST. BBST change the head when needed (B Trees do this mechanism with some other enhancements such as having more than 2 children per node)

Search: O(n)

B-tree

- Ordered
- Each node can have up to *k* children (and *k-1* keys).
- Balanced
- Every node (other than the root) must be at least half full

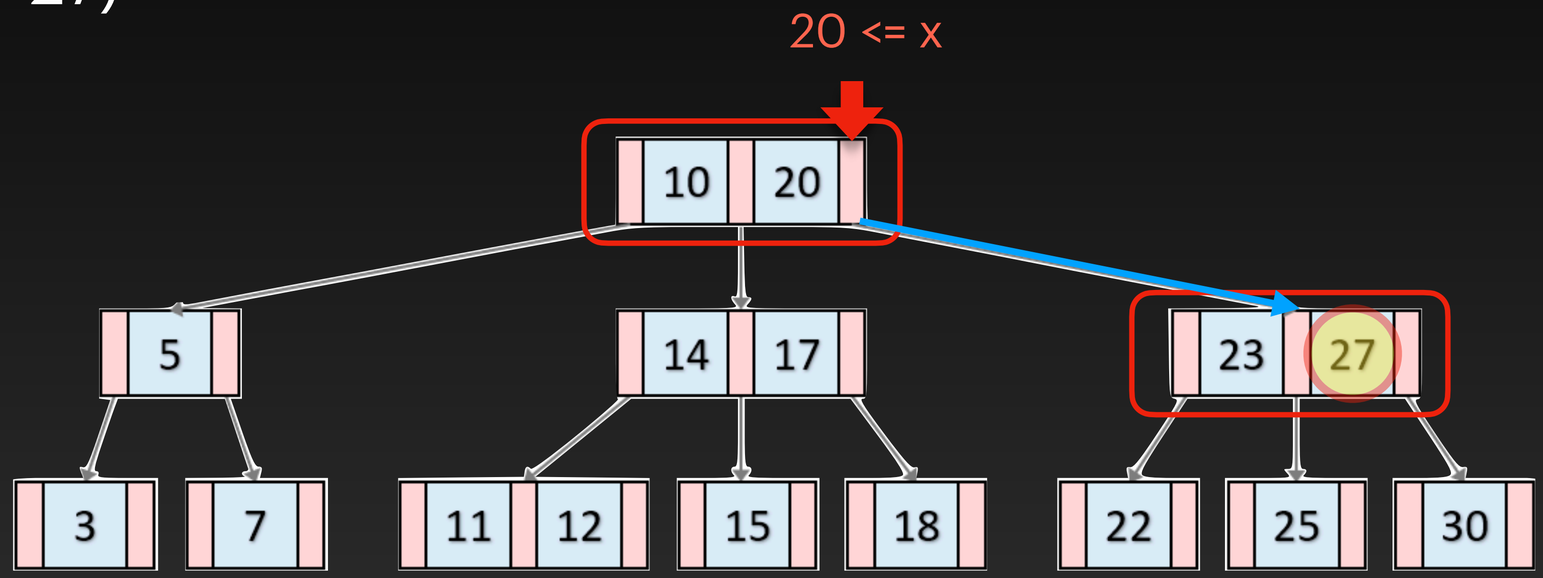
k = 3



- splitting when nodes are expensive in writing and rebalancing the tree but good in reading.
- splitting happens by taking the middle node and put it with the root keys.
- each node have a pointer to the value on disk or row id or actual row

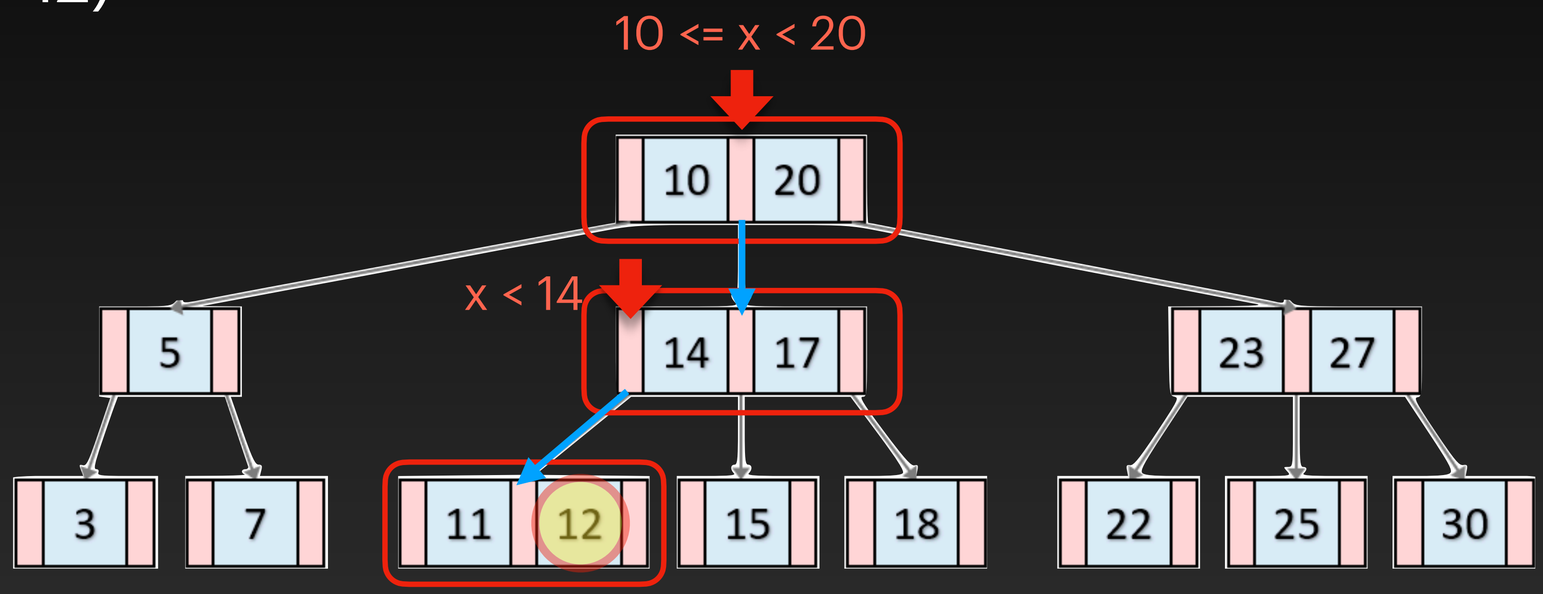
B-tree

Look for (x = 27)



B-tree

Look for (x = 12)

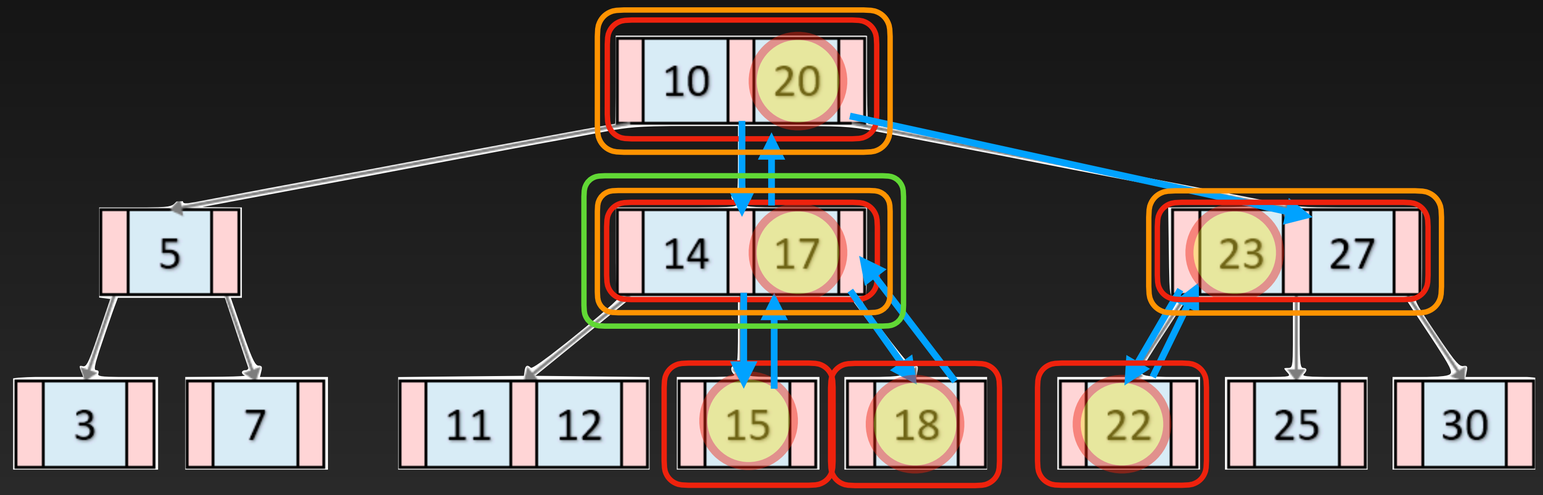


Search: O(log n)

- one page contains multiple pages. so number of loaded pages decreases when using b trees

B-tree

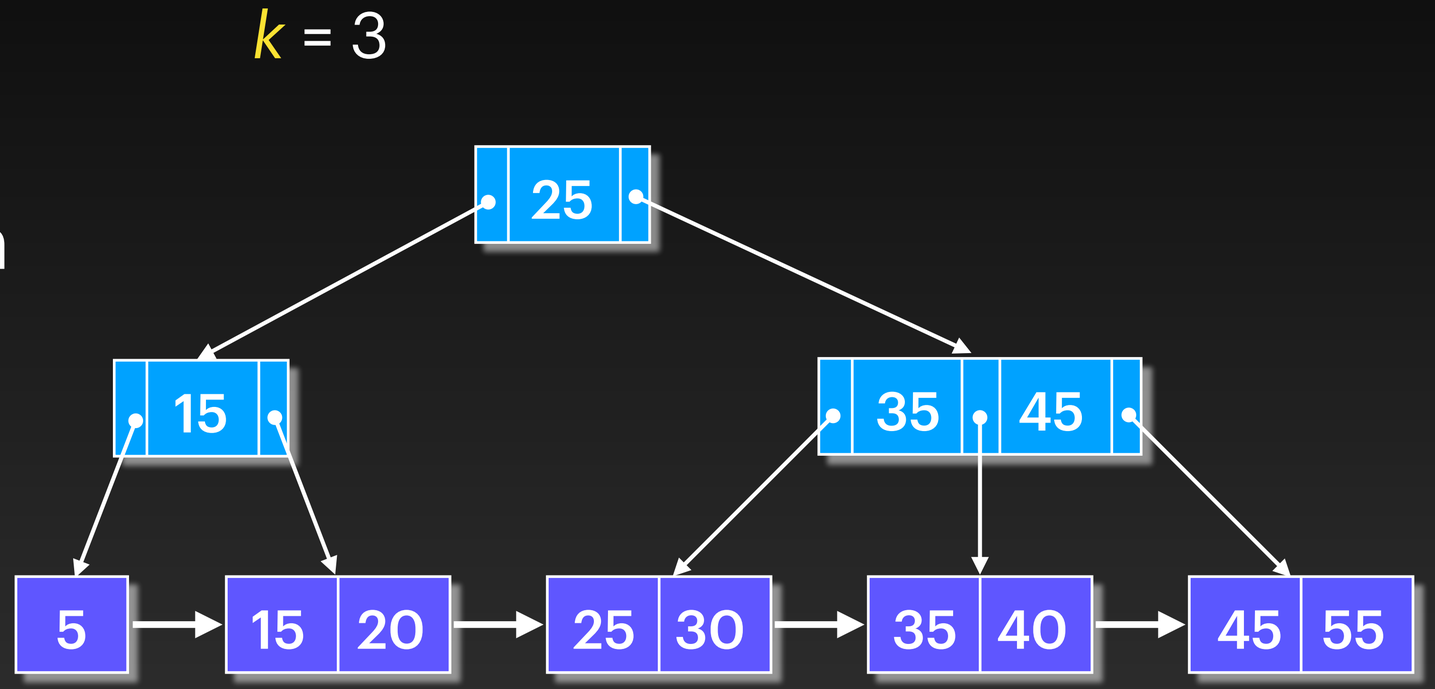
Look for (15 <= x < 24)



Cons:
1. writing cost of rebalancing.
2. pointers take size.
3. range queries are expensive ups and downs and reading same page multiple time.

B+ tree

- Data pointers in the leaf nodes only
- Leaf nodes link to each other (possibly in both directions



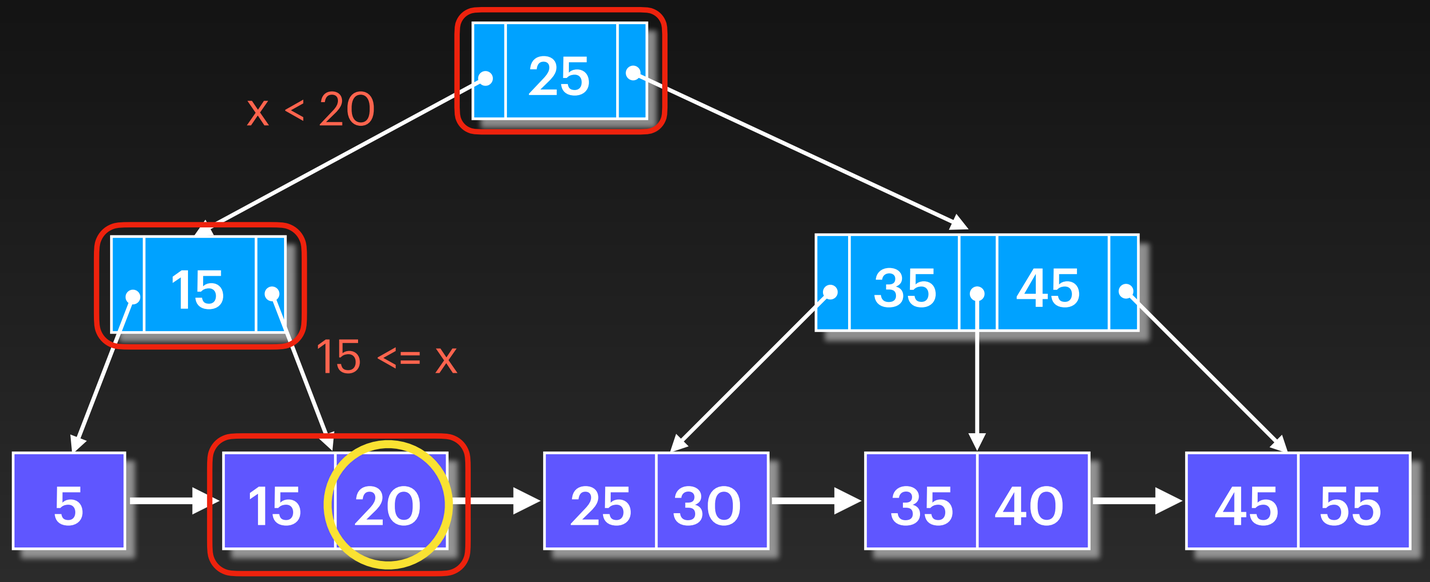
- all nodes are duplicated and sorted in the leaf nodes.

- splitting here is different: it takes the middle leaf node and duplicates it above in the root keys.

- keys are sorted in the leaf with pointers to row data or data itself.
- tree roots are used to find where to reach the keys in leaf.

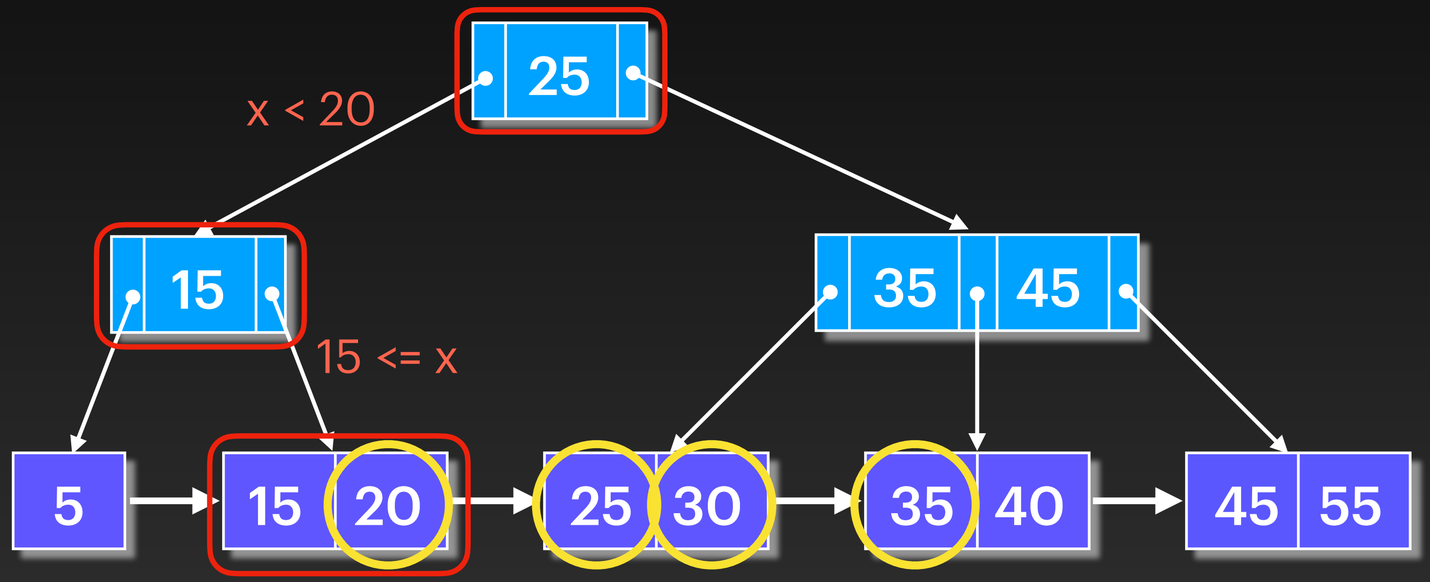
B+ tree

Look for (x = 20)



B+ tree

Look for (20 <= x <= 35)



Pros:
- no pointers in roots which means page can take multiple nodes and number of loaded pages are lesser