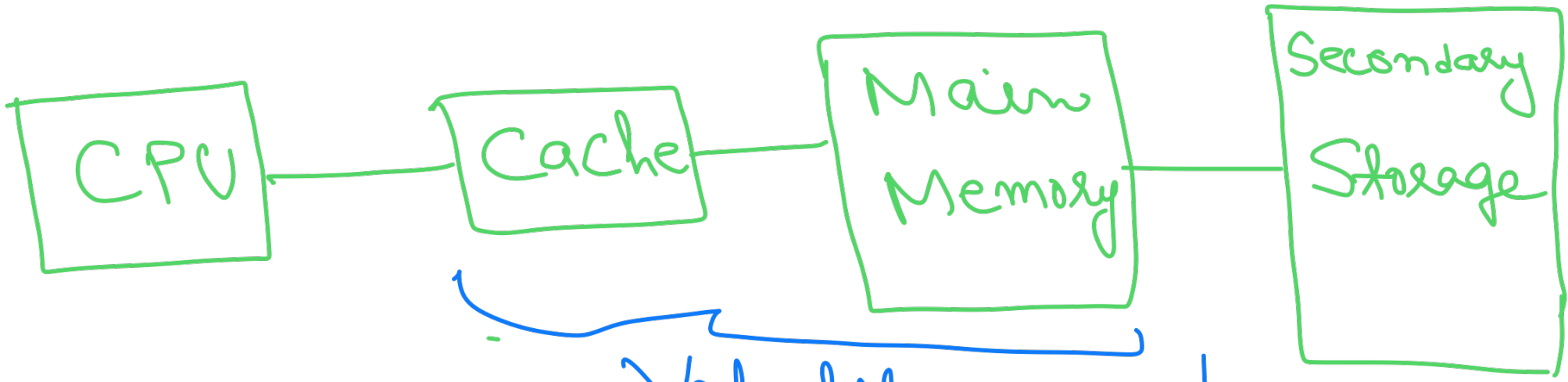
Basic File systems

- Data Dictionary
- Database
- Intermediate Computation
- Queries

Physical Storage Media

- Cache Memory
- Main Memory
- Disk Storage
- Tape Storage

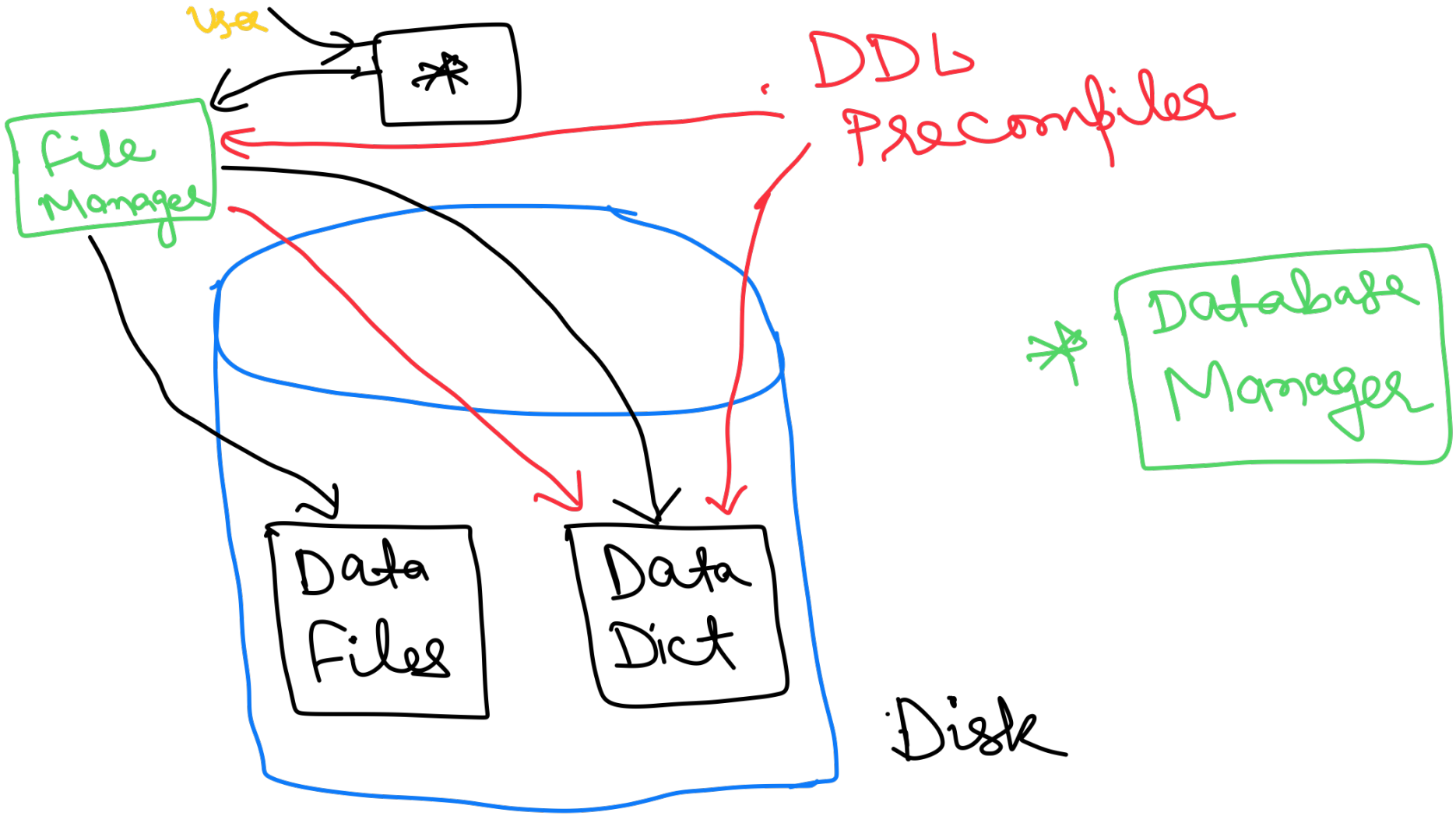


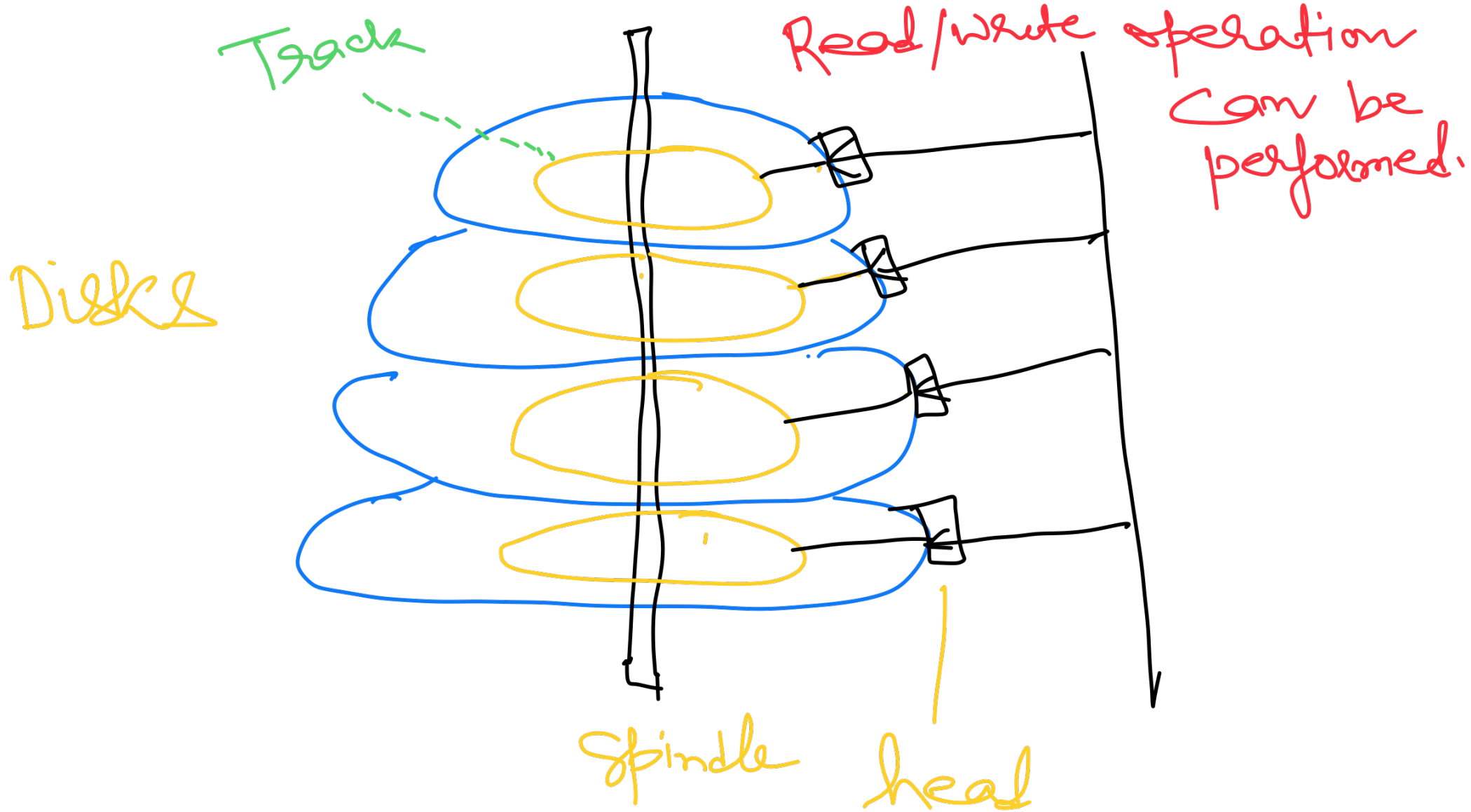
→

- Data Dictionary
- Database
- Intermediate Computation
- Queries

} Non-volatile

} Can/may be performed on Main Memory


User
*
DDL Precompiler
File Manager
Data Files
Data Dict
Disk
*
Database Manager

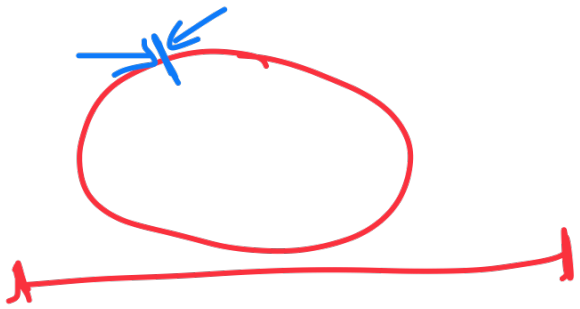

Track
Read/write operation can be performed.
Disks
Spindle
head

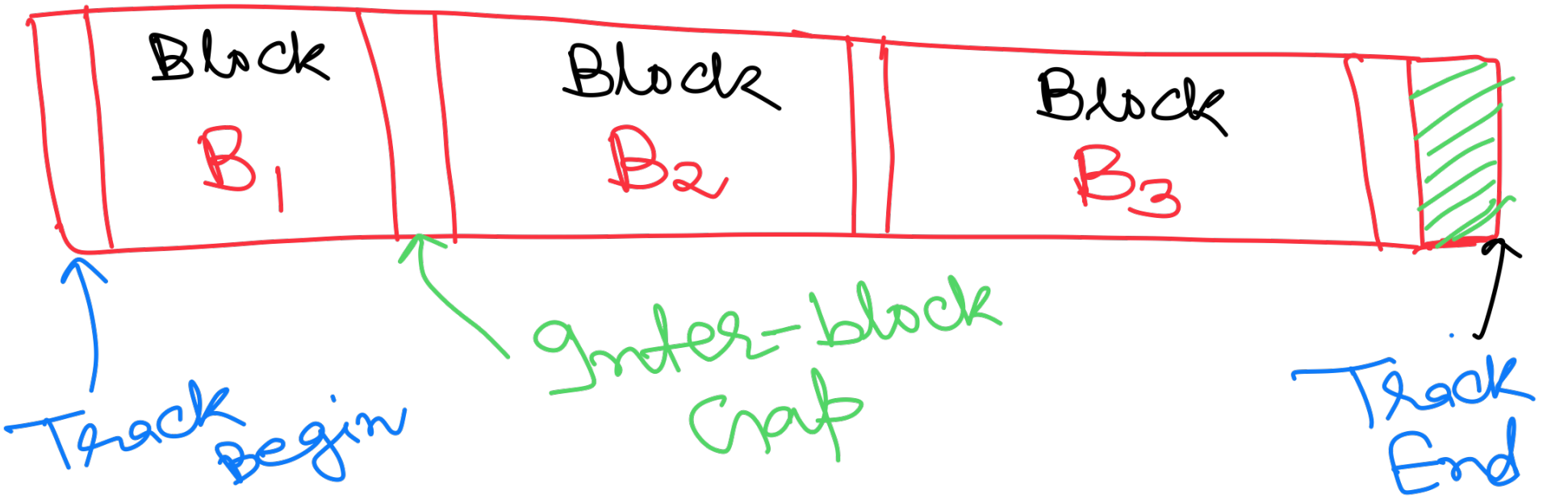
* Disk Storage

- Seek Time
  - → Time required to place the head on the Track
- Latency
  - Latency → Time required to Read Some unit of data.

Disk Organization

- Track (Maybe Cylinder which contain multiple tracks)
- Blocks
- Records
- Files





→ In hard disk, Read/Write operation happened blockwise.

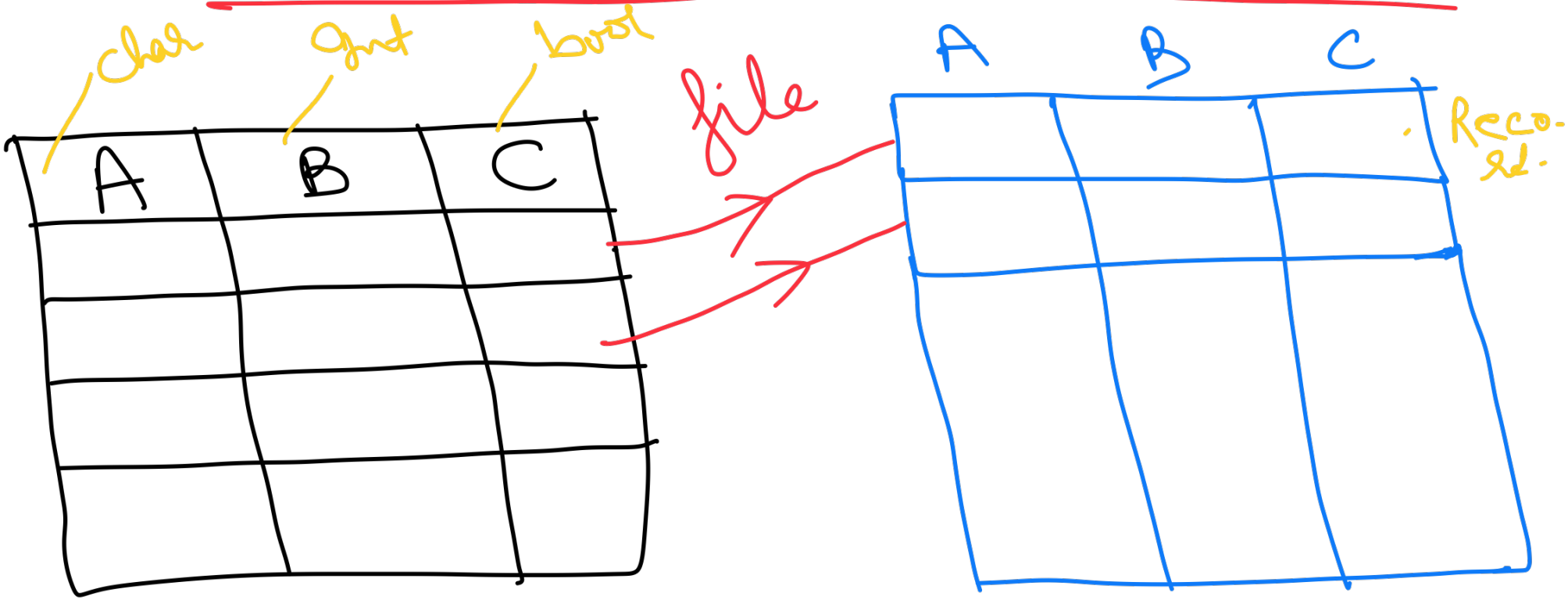
However, in main memory, Read/Write operation happened bytewise.

One block contains multiple bytes.

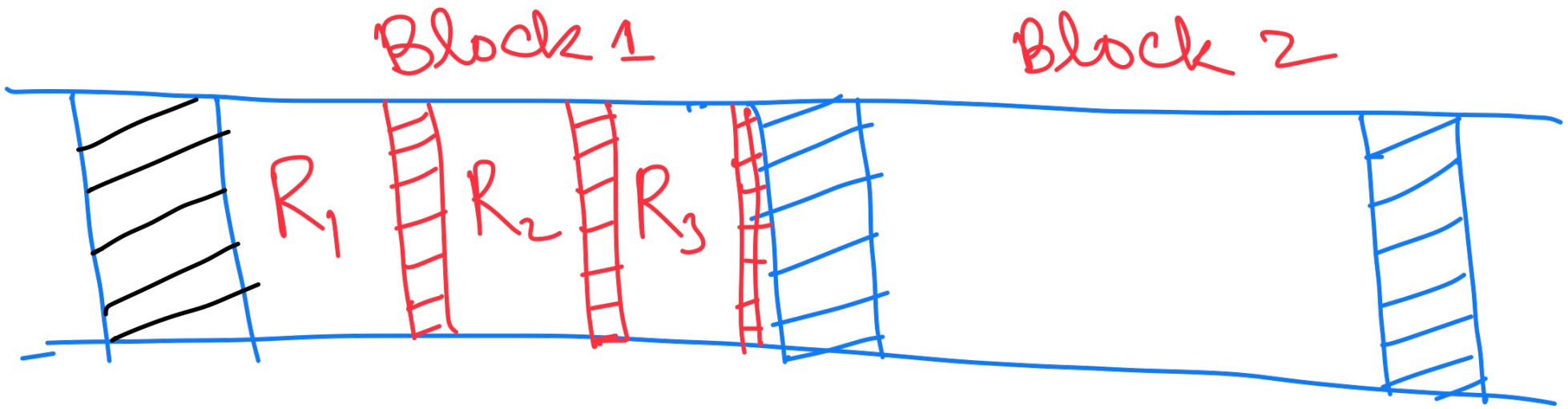
Records in Blocks

* fixed length blocking for fixed length records
* Variable length Unspanned Records
* Variable length Spanned Records

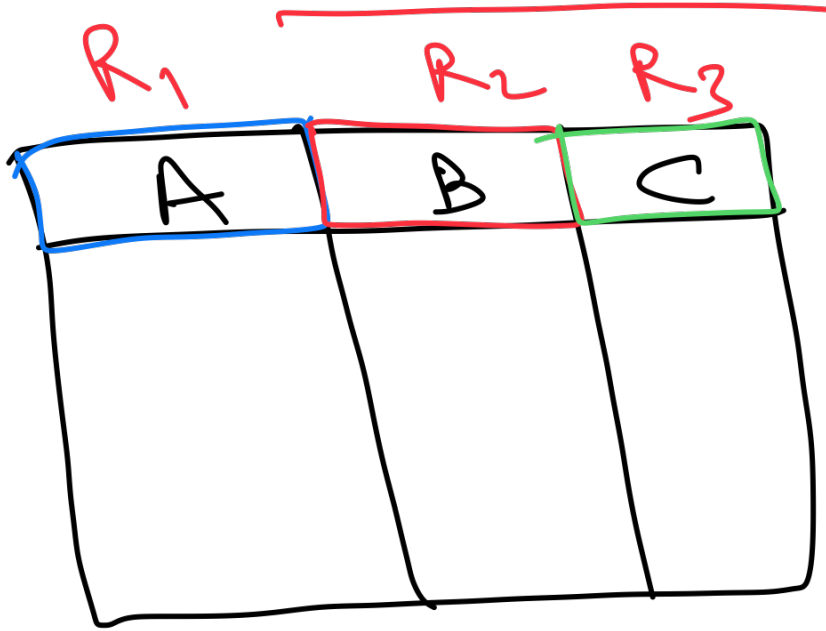
Fixed Length Records

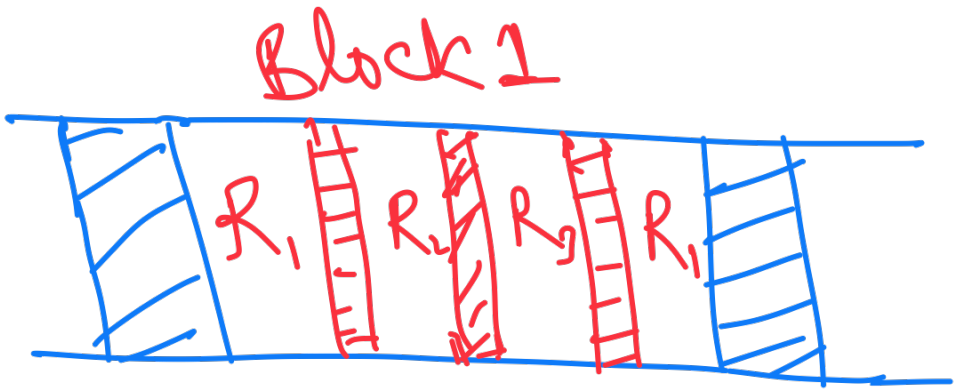


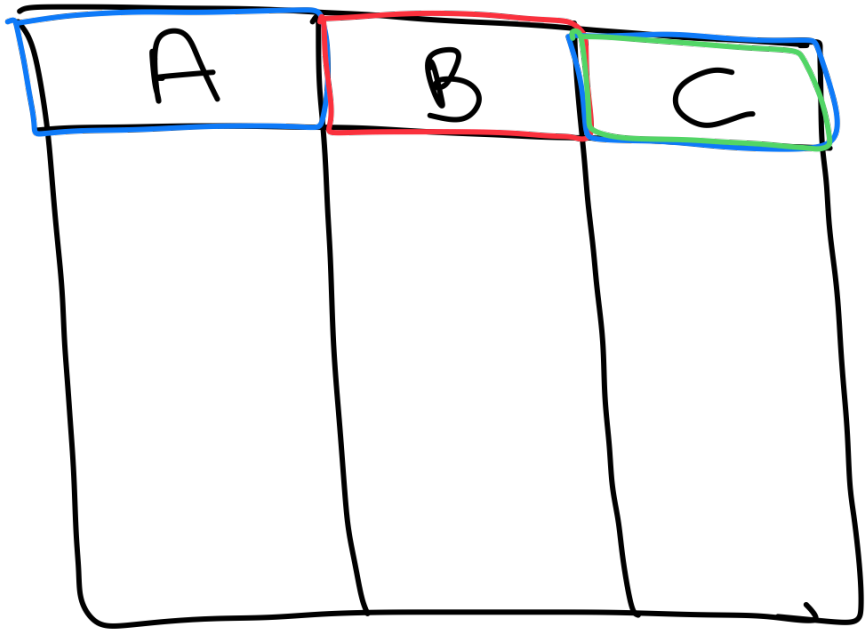
One Table per file



Variable length Records (VLR)



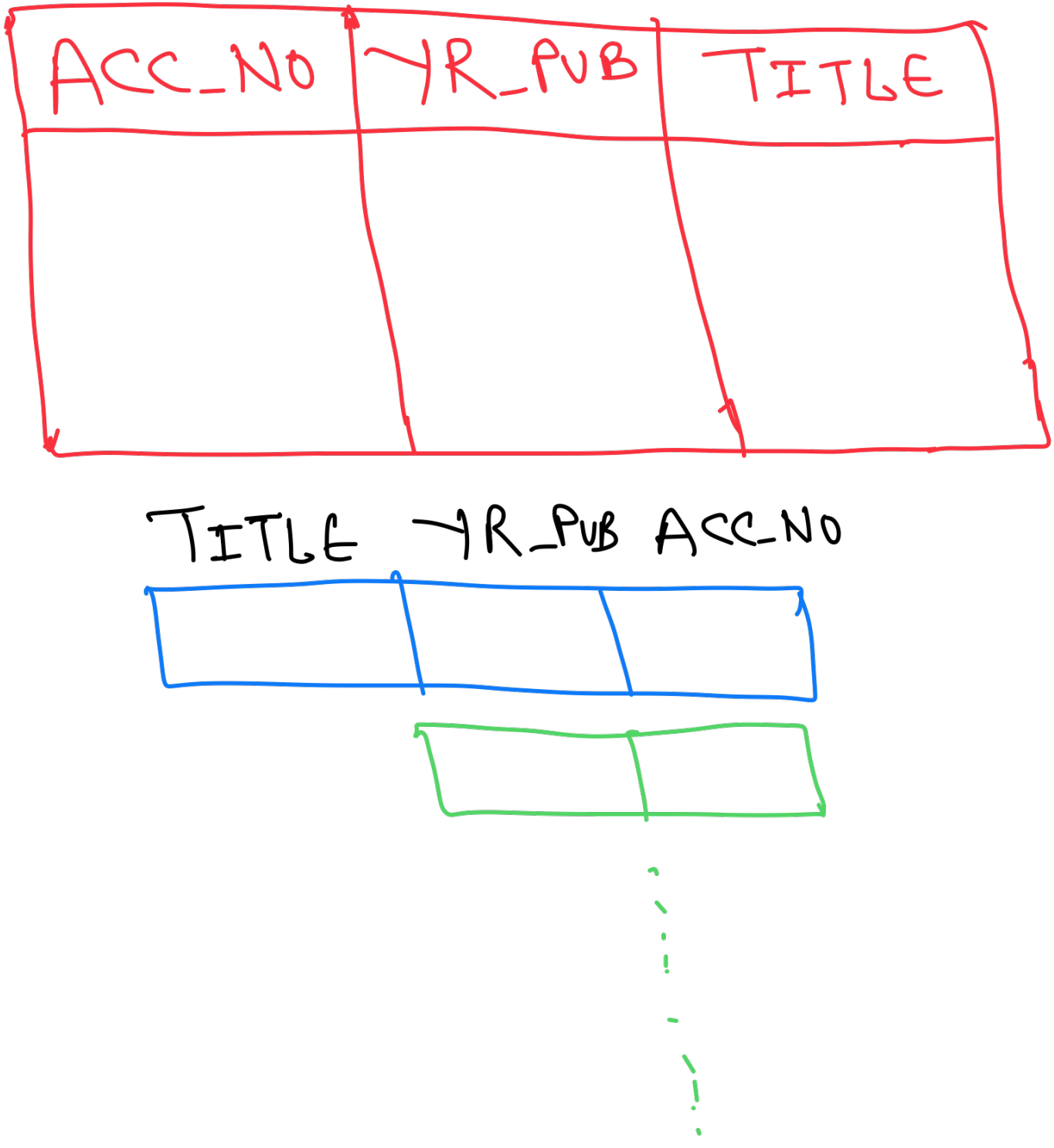




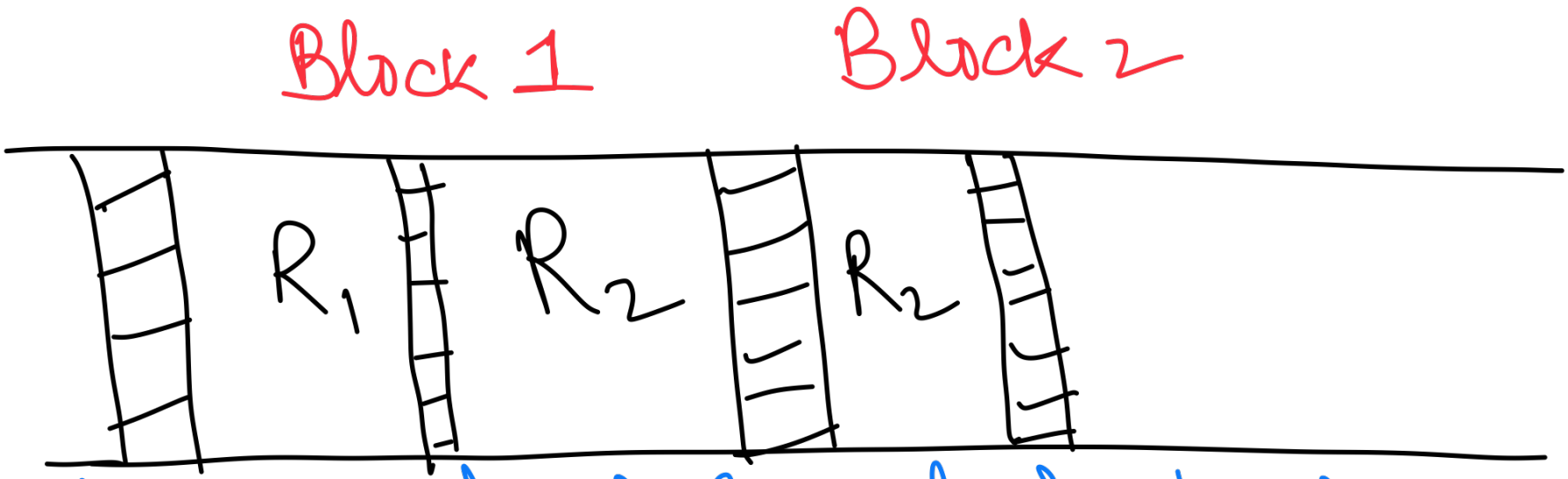
In one file, we may store more than one table

→ Unspanned type VLR — It does not allow same record to be part of more than one block.

→ Spanned type VLR — It allows.


ACC_NO
YR_PUB
TITLE
TITLE YR_PUB ACC_NO

Variable length Spanned Records



We allow part of Record to be in one block and rest of it to be on the next block.

Pinned and Unpinned Records

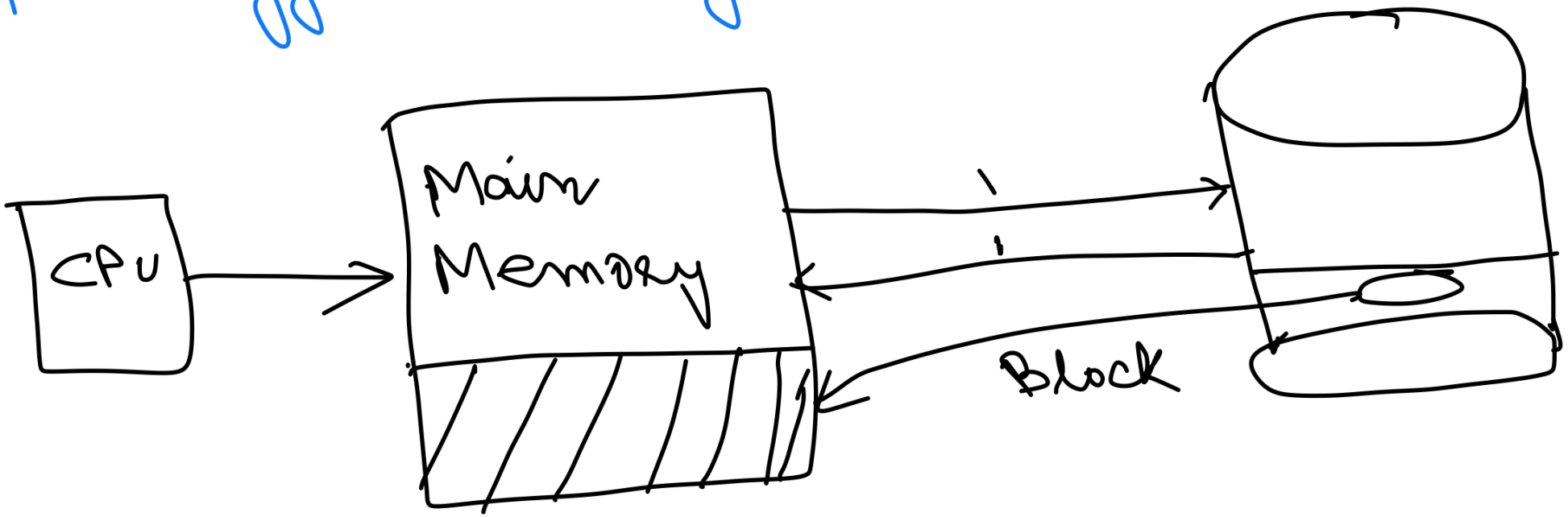
Files

* Data Dictionaries

- Relation Names
- Attribute Names, Domains
- View Definitions
- Integrity Constraints
- Access Rights
- Corresponding Data files

* Data files
  - One relation one file
  - One relation many files
  - Many relation one file

* Buffer Management



→ Locality of Reference

→ Block Replacement Policy

→ Pinned and Unpinned Records

File Organization

- Heap or pile
- Sequential files
- Indexed sequential files
- Hashed files
- B^+-Trees

A file refers to a collection of related records, or data, which are stored and organized logically.

Heap File (File)

very easy

Insertion : By stacking

Deletion : By setting a field in Record

very costly

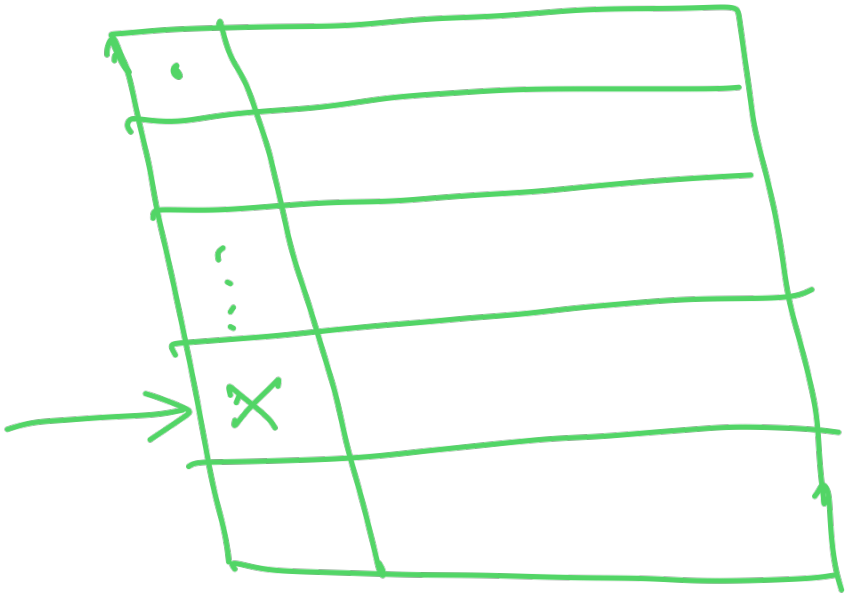
Search : Brute Force

Garbage Collection

Reorganize the file (Remove the Deleted entries)

Sequential files

Records are kept in sequential order on some attributes (s) – usually the primary key



5721	Compiler Design	1978
5732	Database Engg	1986
5734	Database Engg	1986
5735	Algorithms	1992
5740	Graphics	1997

5733	Graph Theory	1976

Insert
Delete
Search / Find

Block Search →

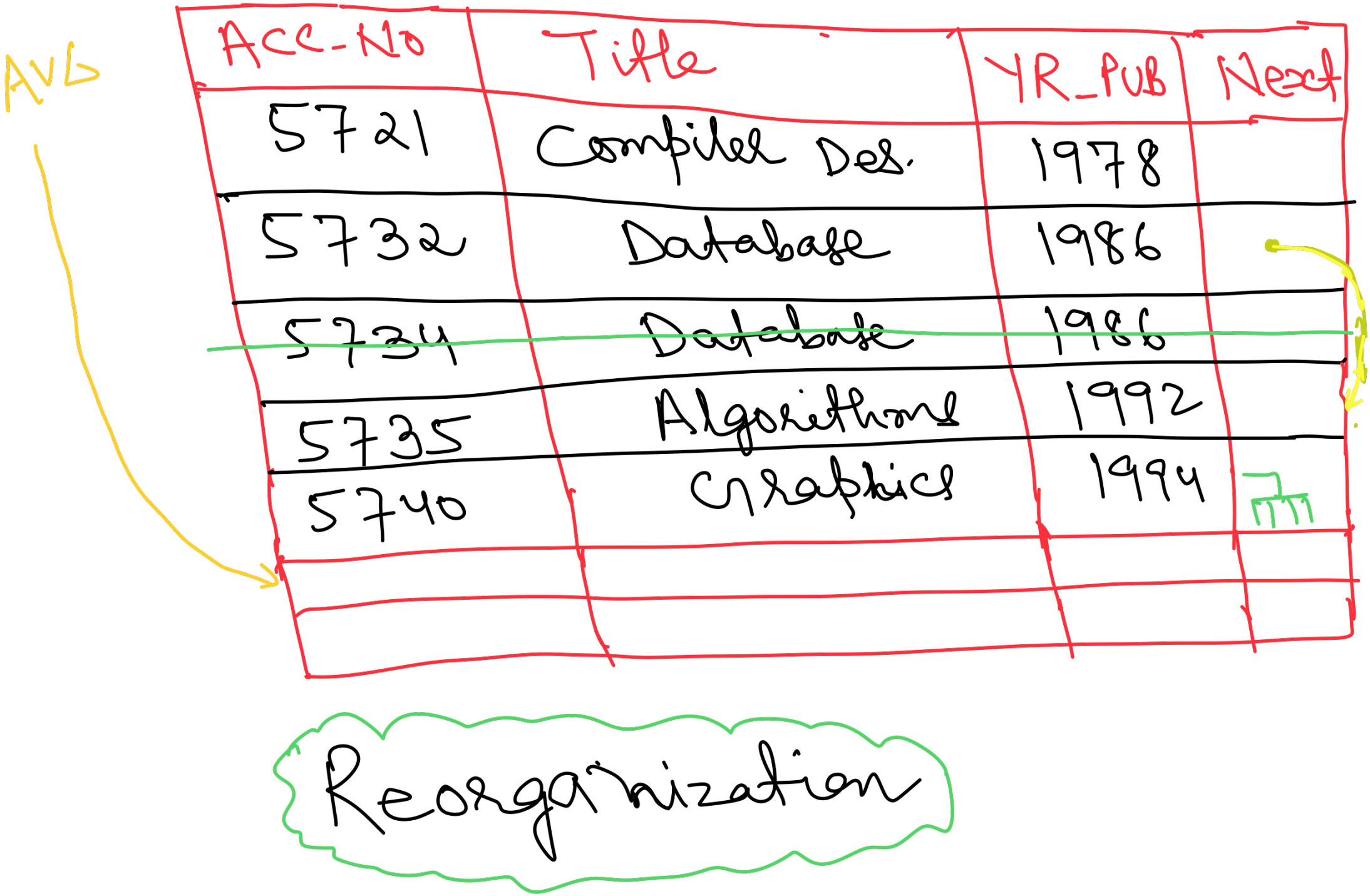
Block search

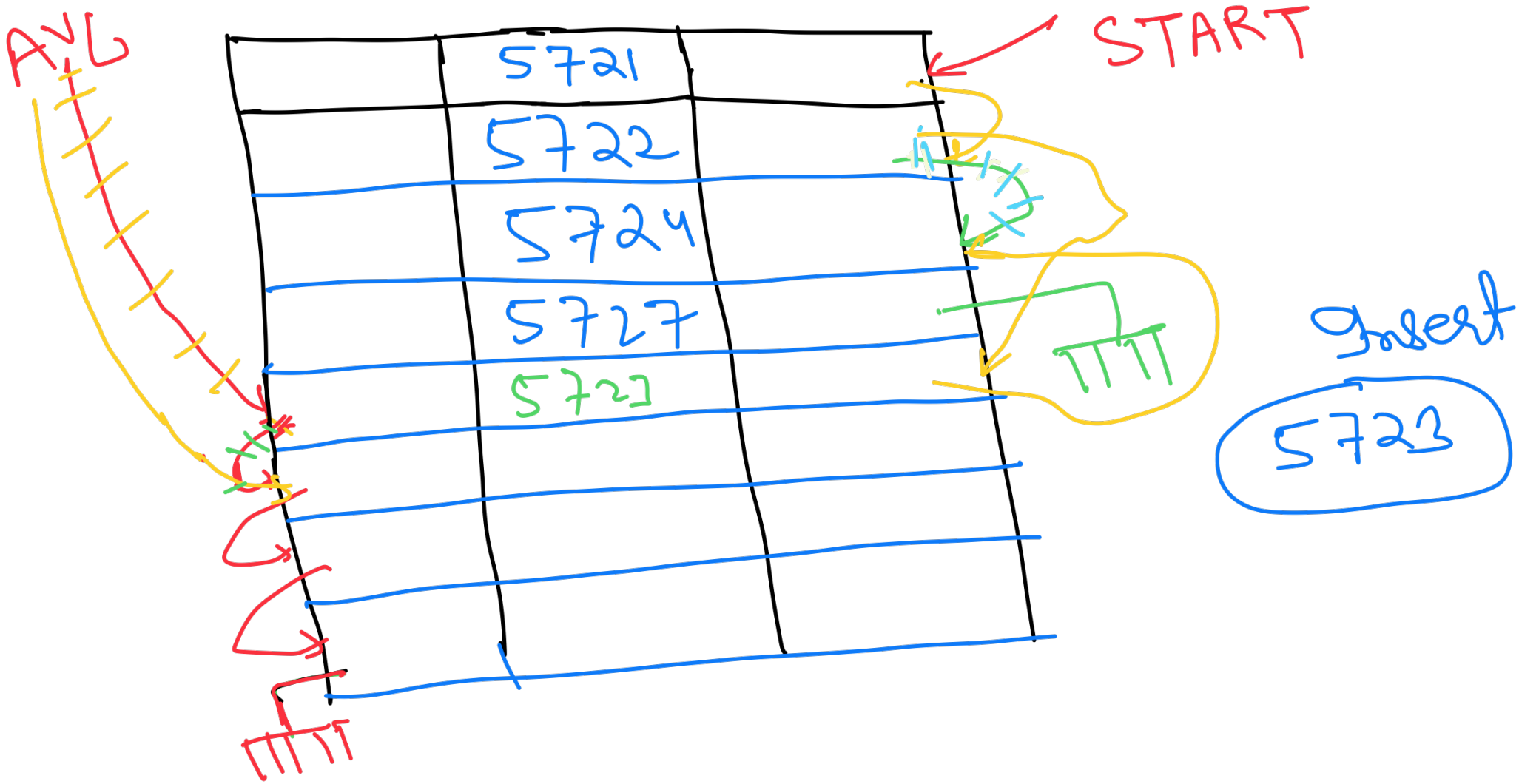
Gets a block

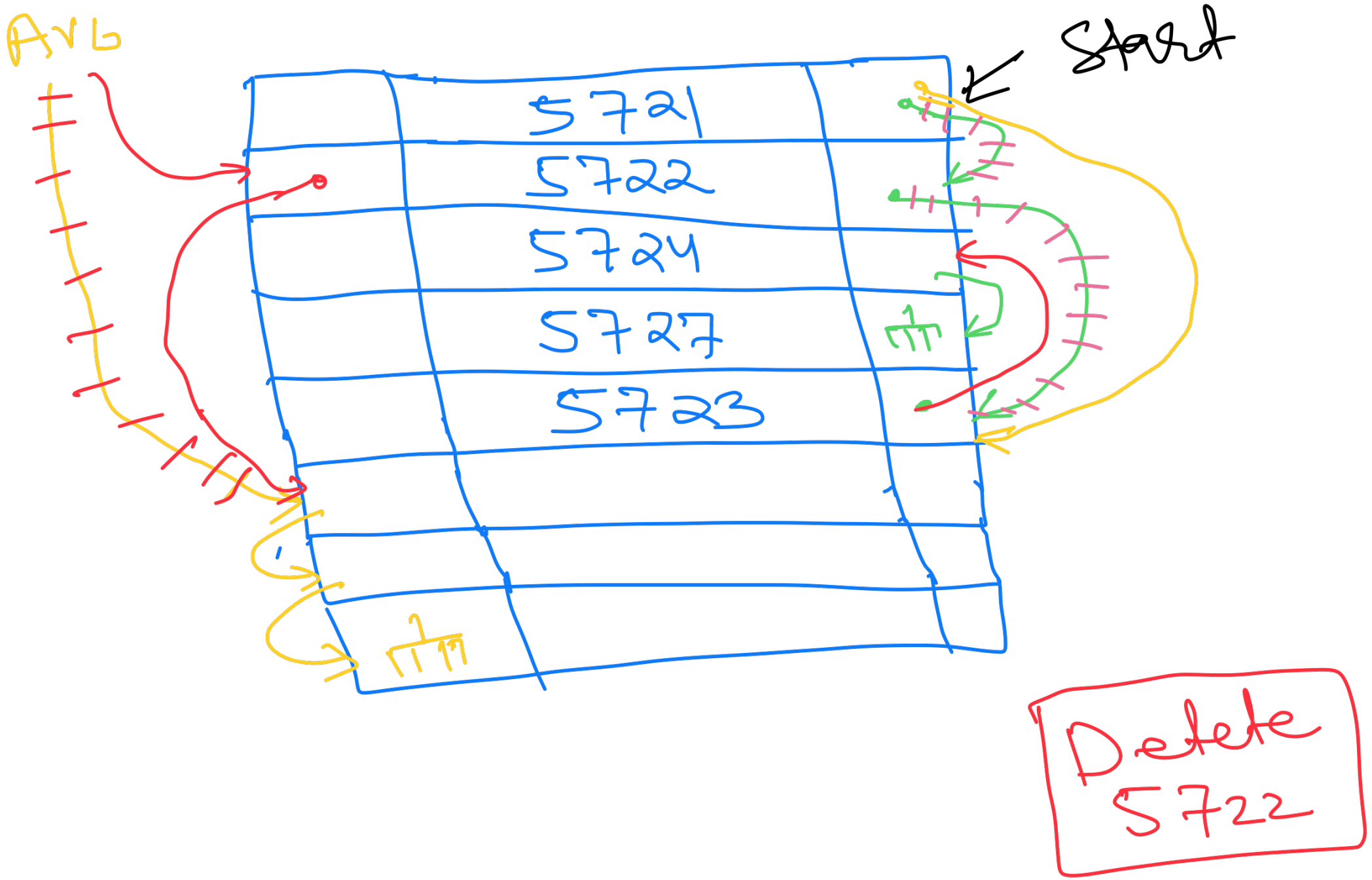
Binary search in the block

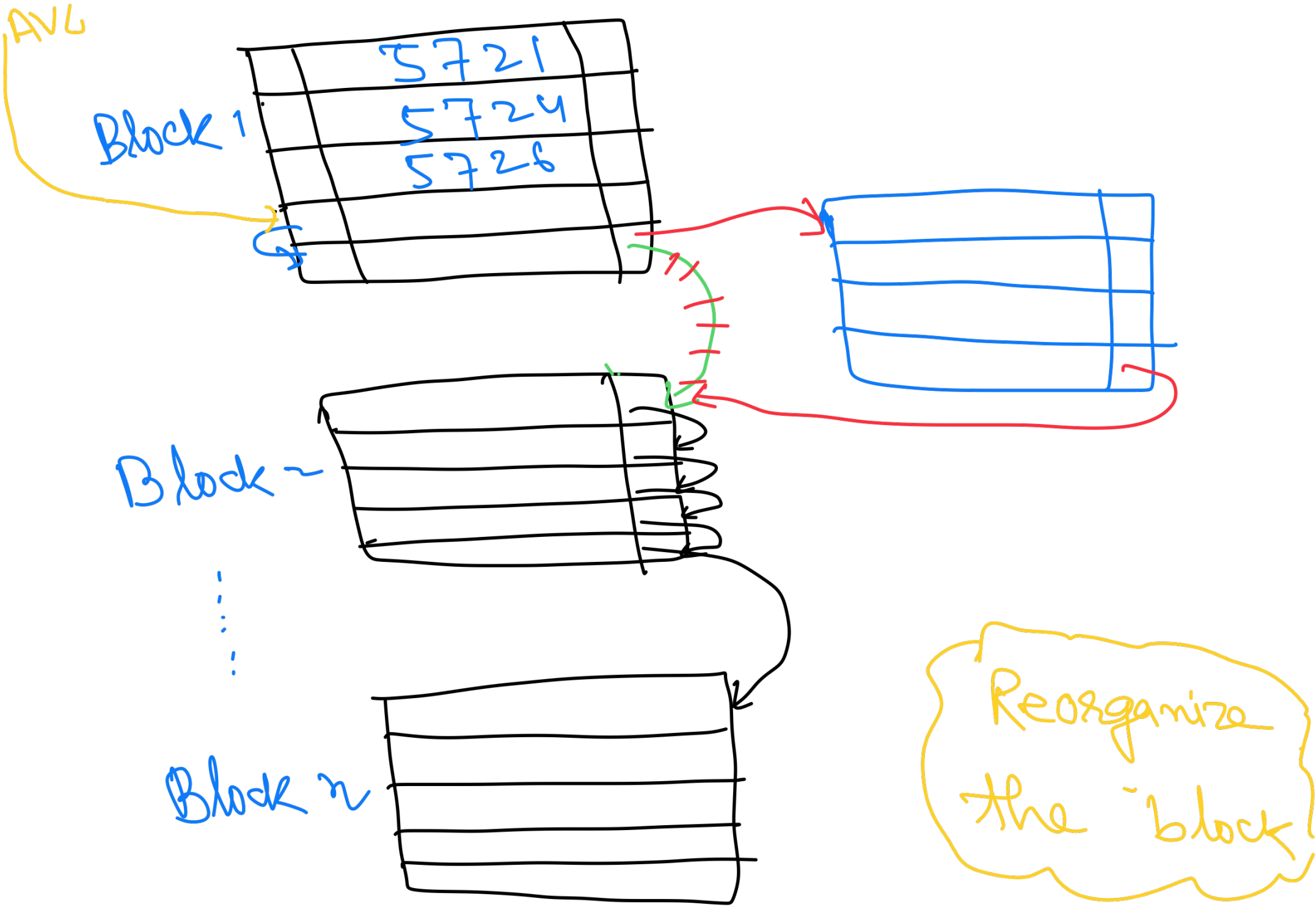
Acc_No	Title	Yr_Pub	NEXT
5721	Compiler Des.	1978	
5732	Database	1986	
5734	Database	1986	
5735	Algorithms	1992	
5740	Graphics	1994	7th
5733	Graphics	1976	

Insert


AVL
ACC-No
Title
YR_PUB
Next
5721
Compiler Des.
1978
5732
Database
1986
5734
Database
1986
5735
Algorithms
1992
5740
Graphics
1994
Reorganization


AVL
START
5721
5722
5724
5727
5723
Insert
5723


AVL
Start
5721
5722
5724
5727
5723
Delete
5722


AVL
Block 1
5721
5724
5726
Block ~
Block n
Reorganize
the block

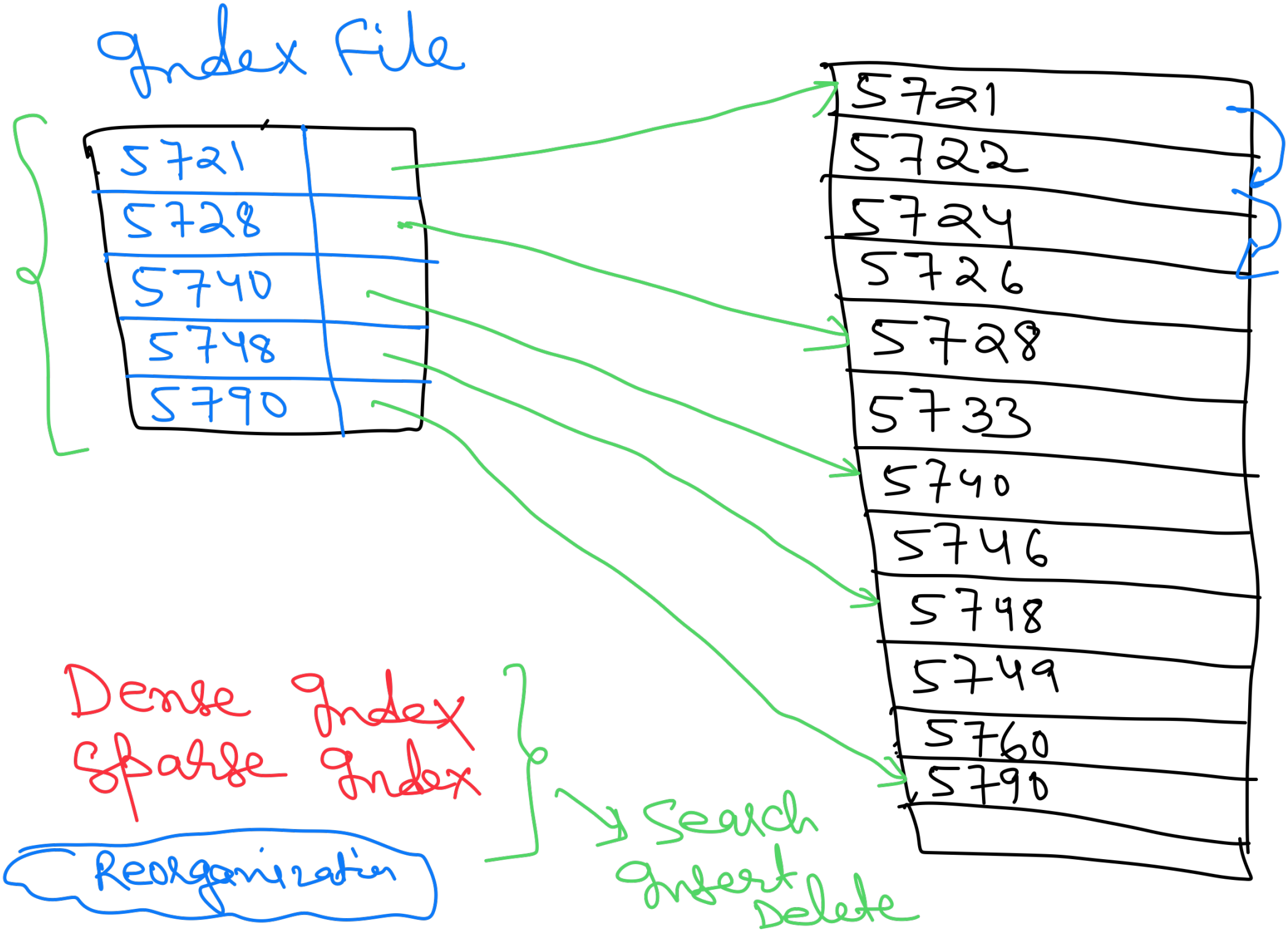
→ Sequential access in Sequential file, hence it is making process slow.

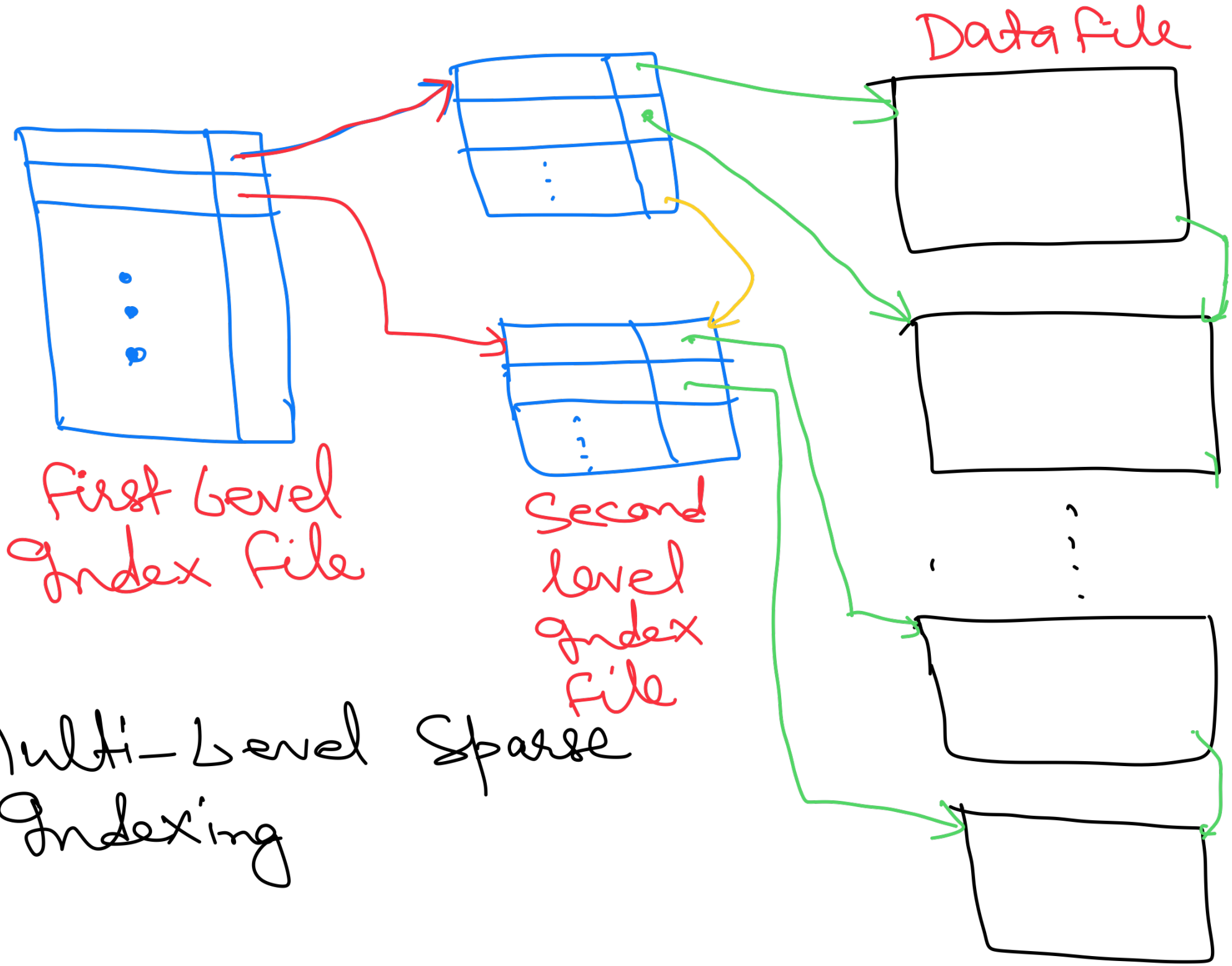
Indexed Sequential files

Maintain an index over the SEQ file.

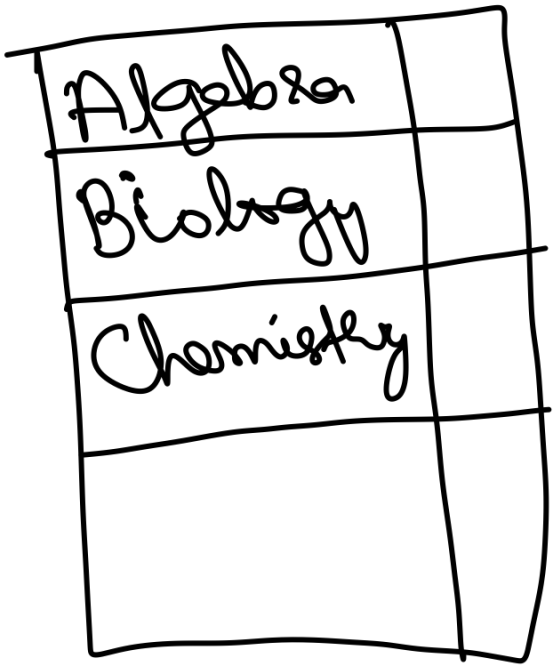
↓

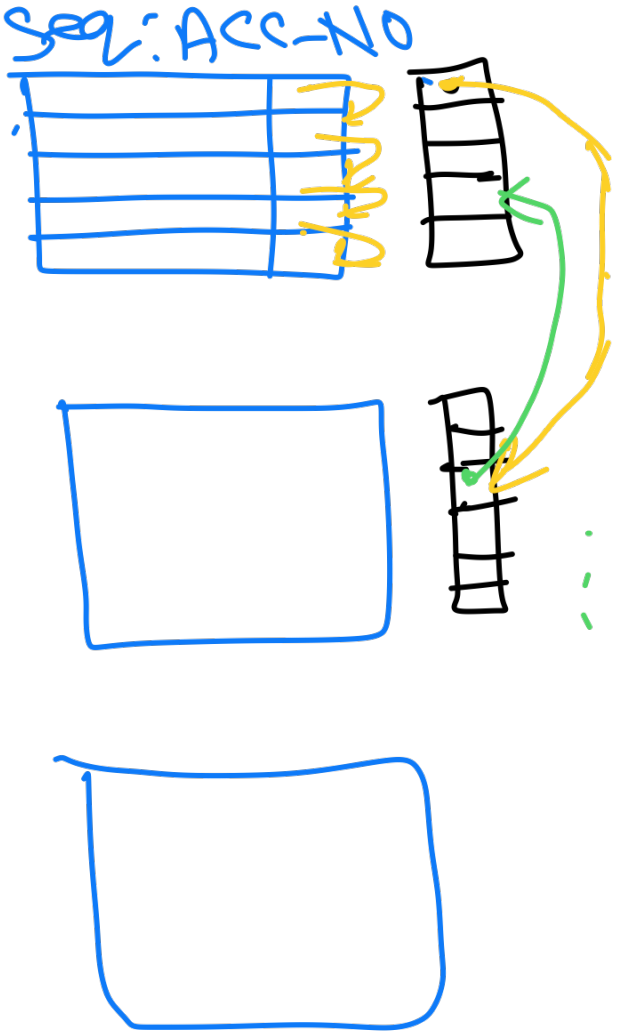
Smaller file


Index File
5721
5728
5740
5748
5790
5721
5722
5724
5726
5728
5733
5740
5746
5748
5749
5760
5790
Dense Index
Sparse Index
Reorganization
Search
Insert
Delete

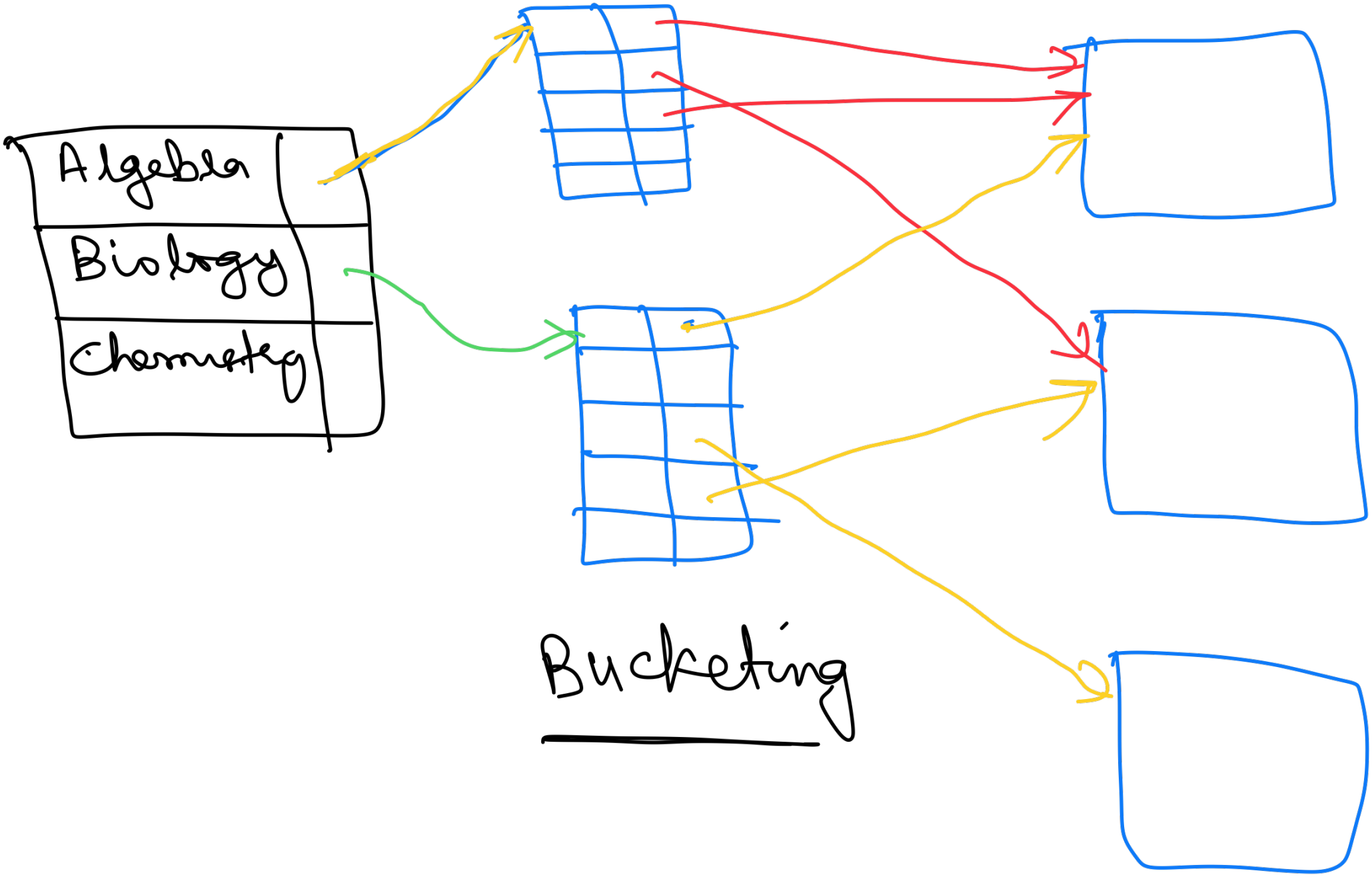


Multi-Level Sparse Indexing



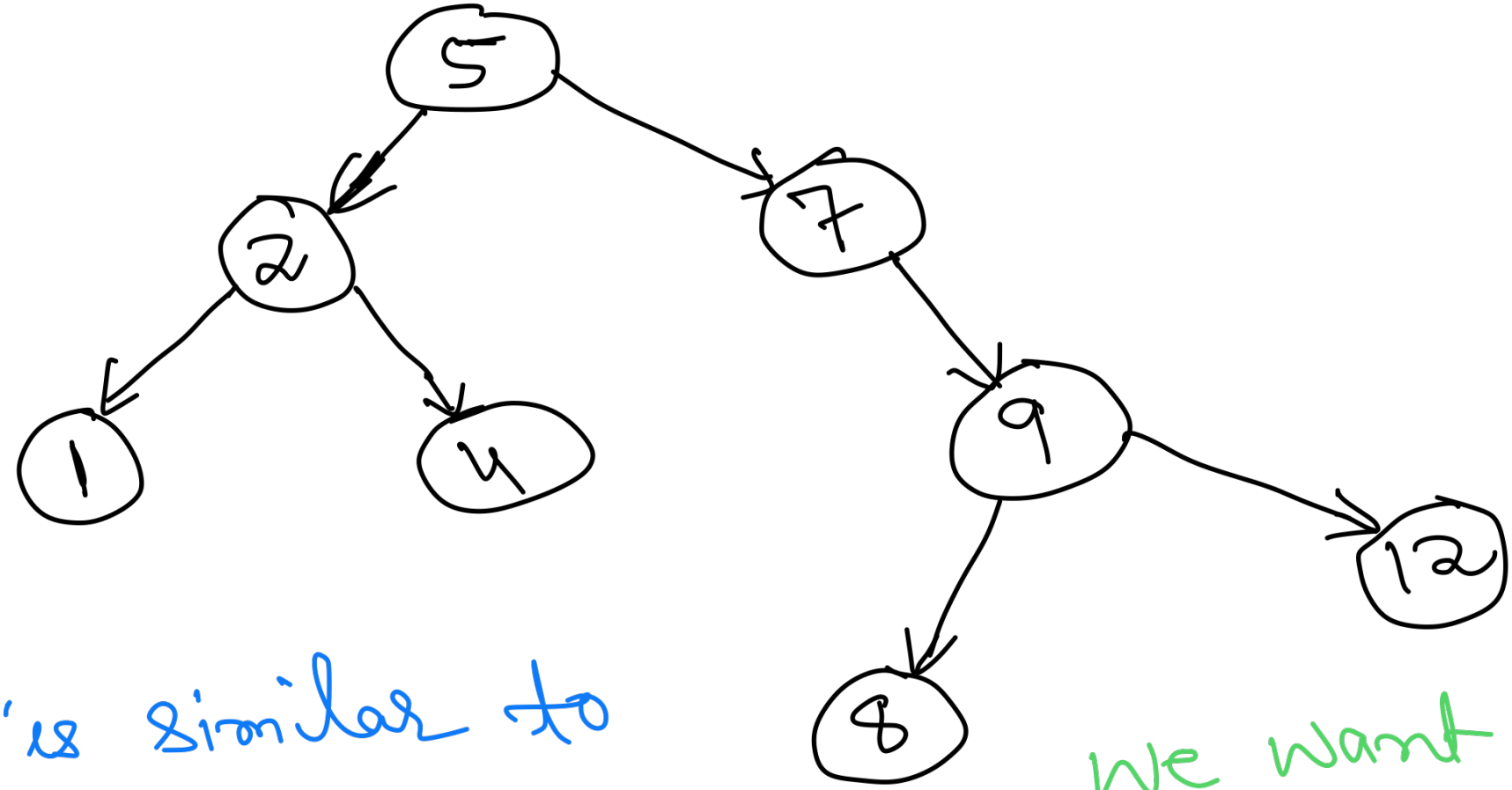


Secondary Index



Bucketing

Binary Search Tree (BST)

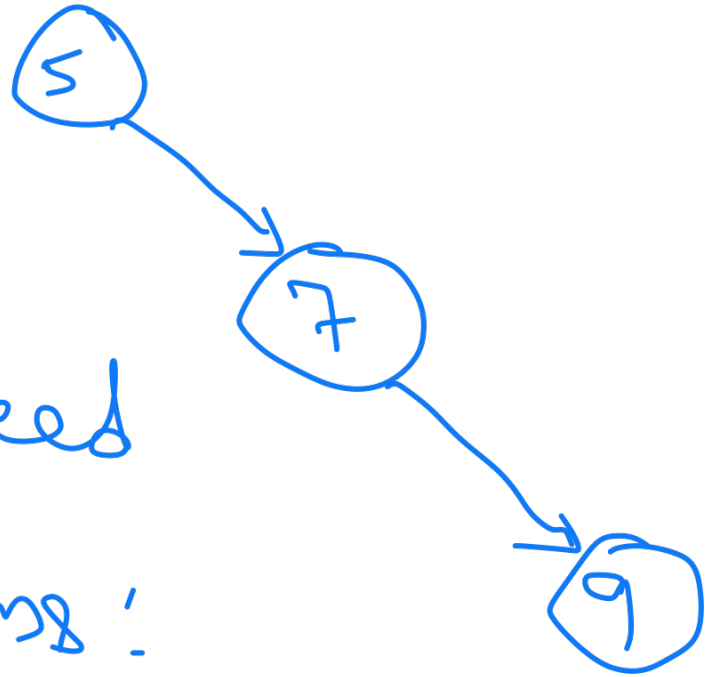


It is similar to multi-level indexing.

We want this BST to be balanced.

- n elements, height $\leq \log_2 n$

→ BST would be balanced when for n elements, the height of this BST should not be more than $\log_2 n$.



not balanced

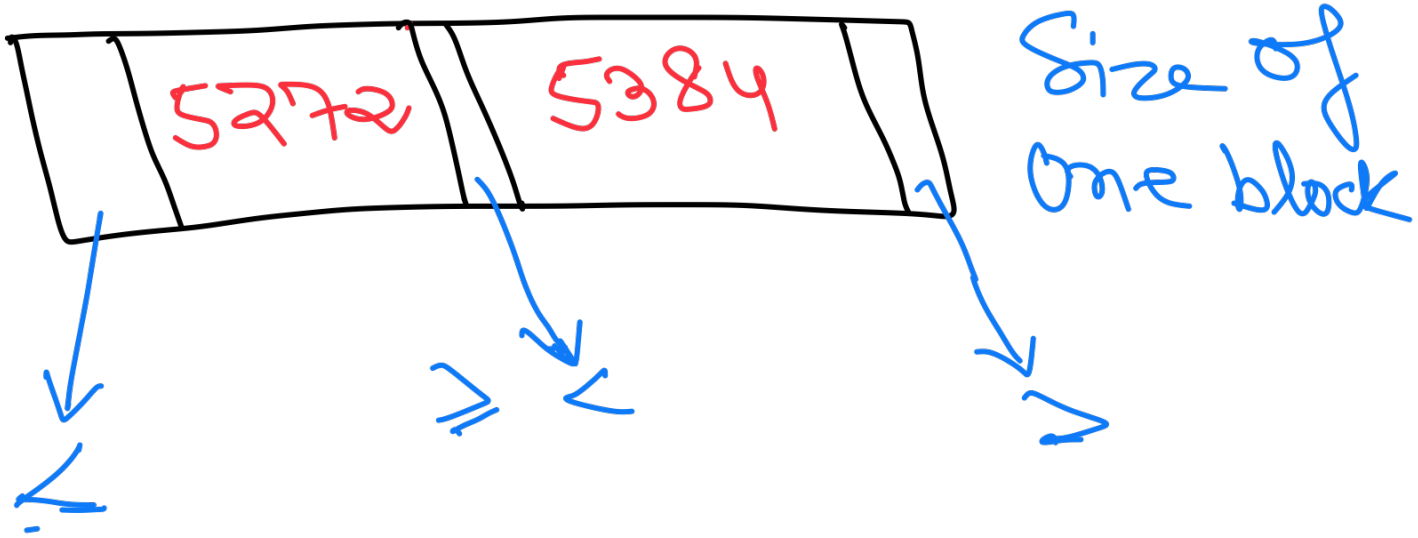
Balancing Algorithms:

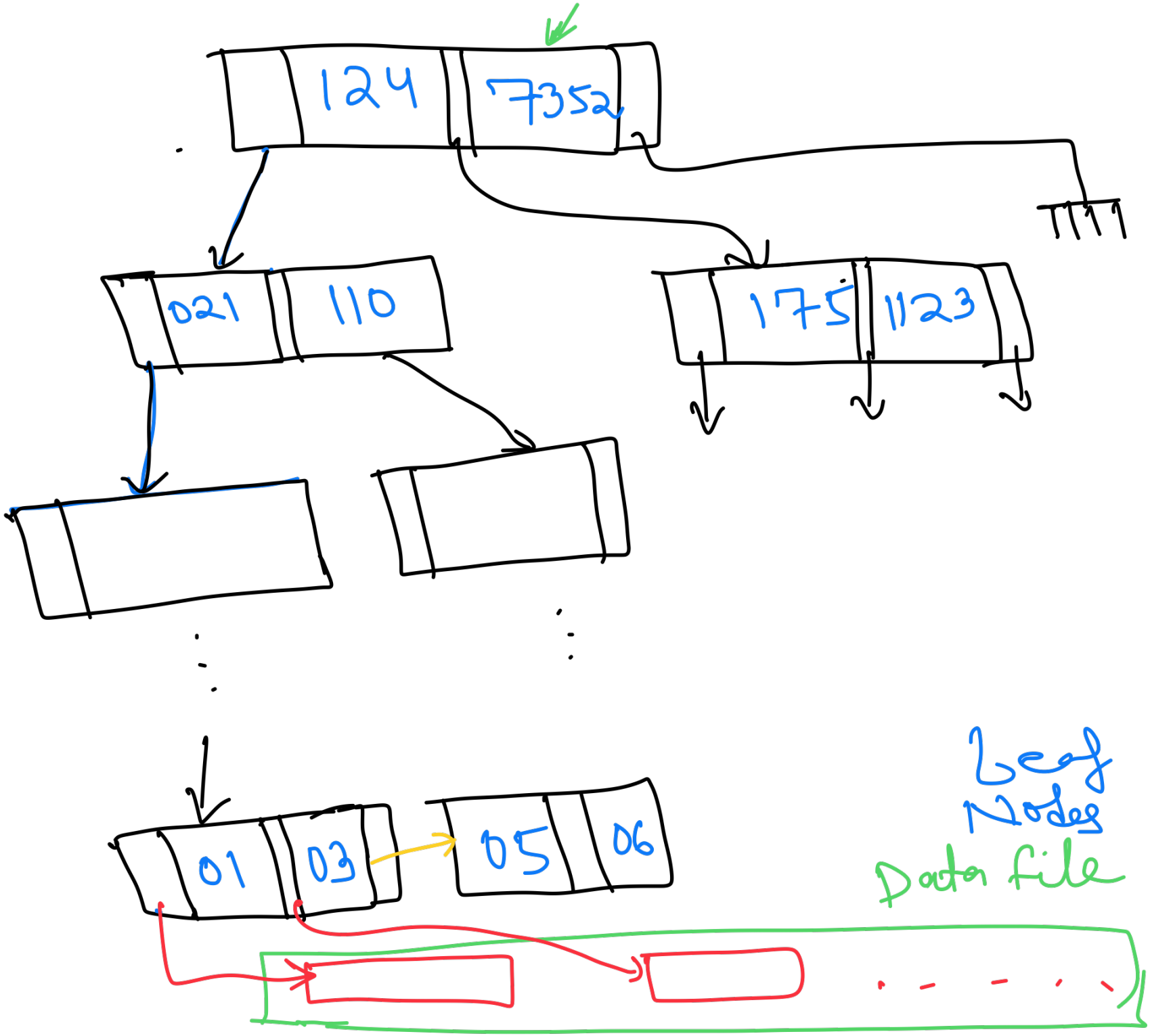
AVL, B-Tree, B+ Tree

B^+-Tree Index files

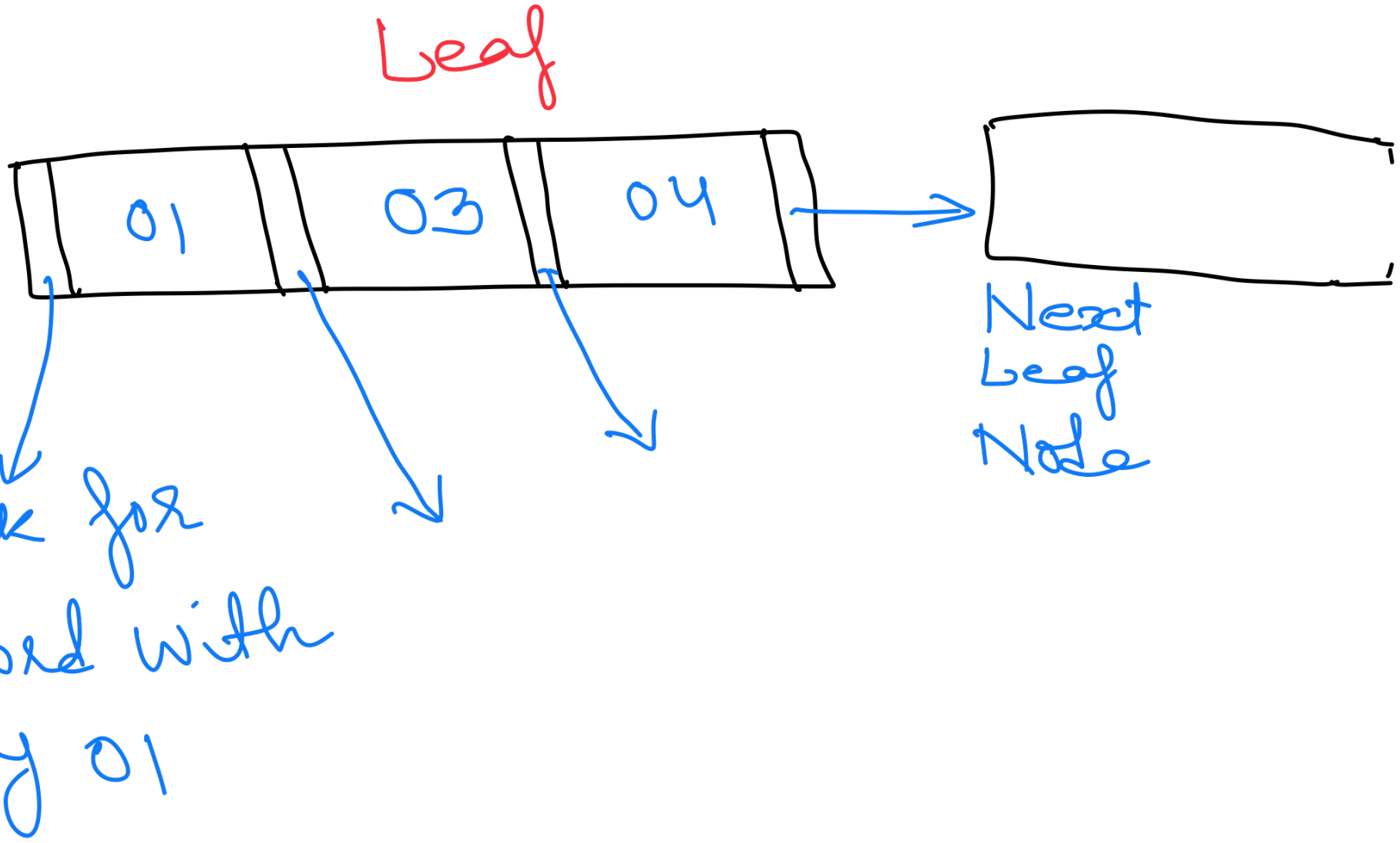
K Records : Each record corresponds to a key

K+1 Pointers

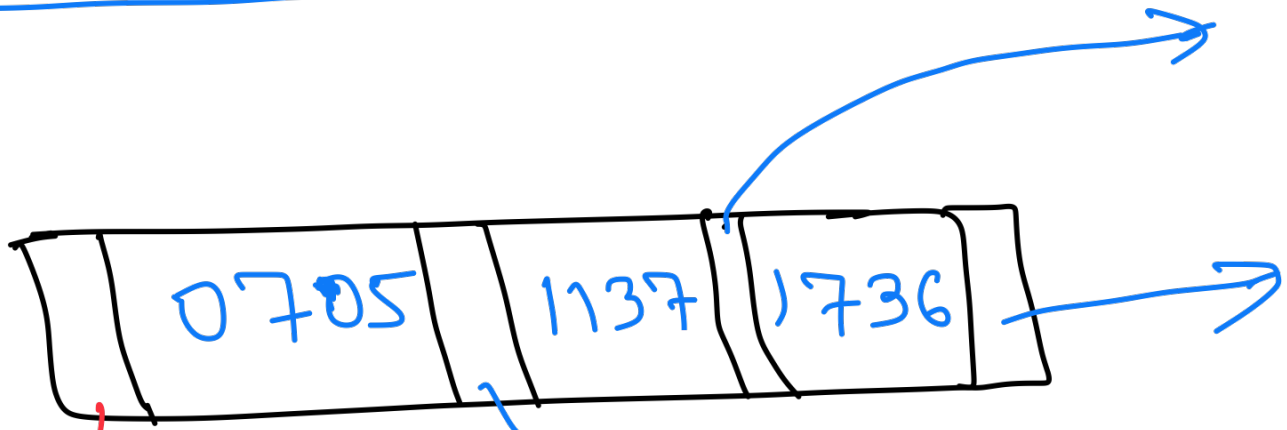



124
7352
021
110
175
1123
01
03
05
06
Leaf Nodes
Data file

#


Leaf
01
03
04
Next
Leaf
Node
Block for
Record with
Key 01

Non-leaf Node



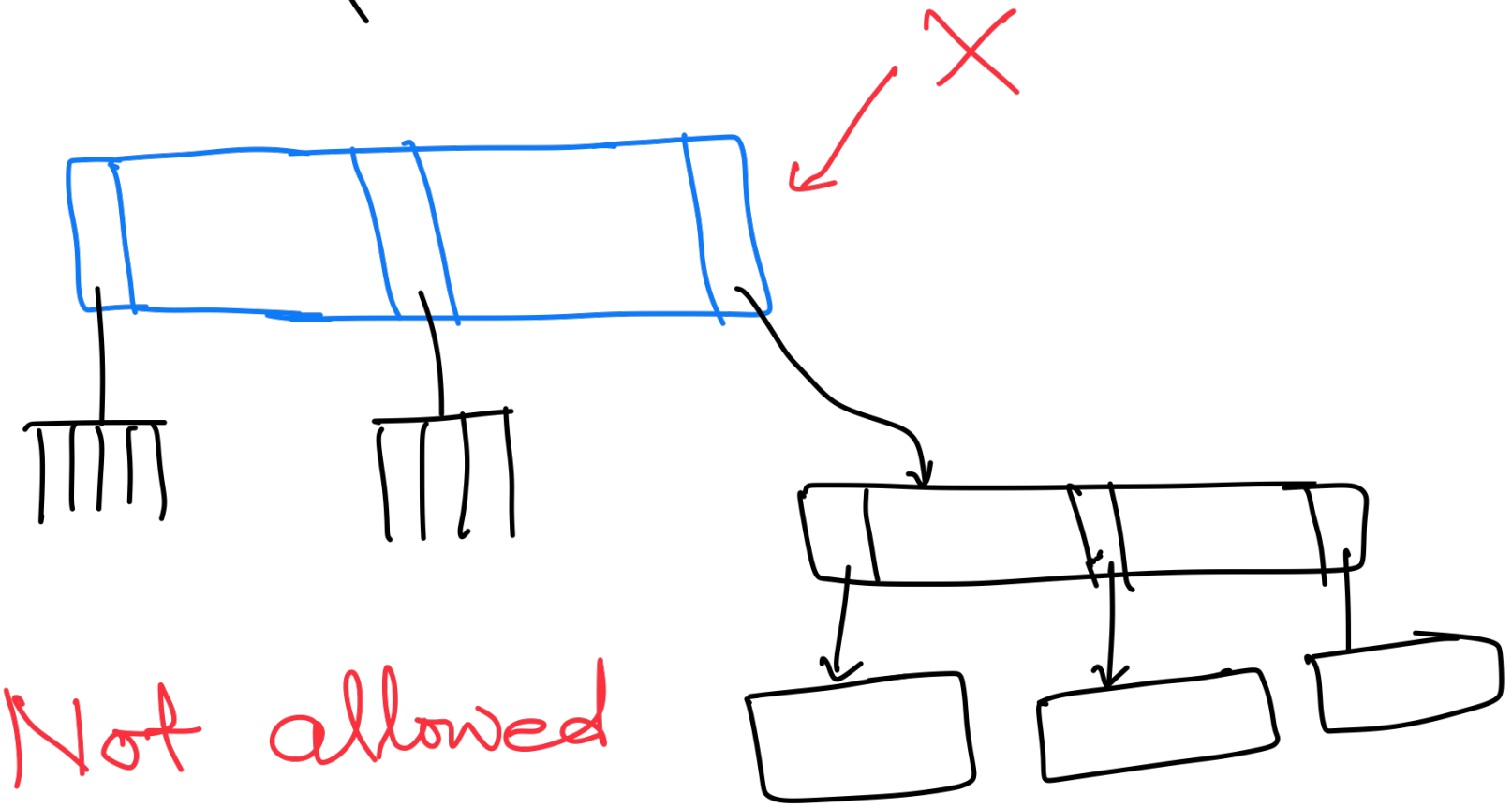
Index block
(Leaf or Non-leaf)

- for records with
  key < 0705

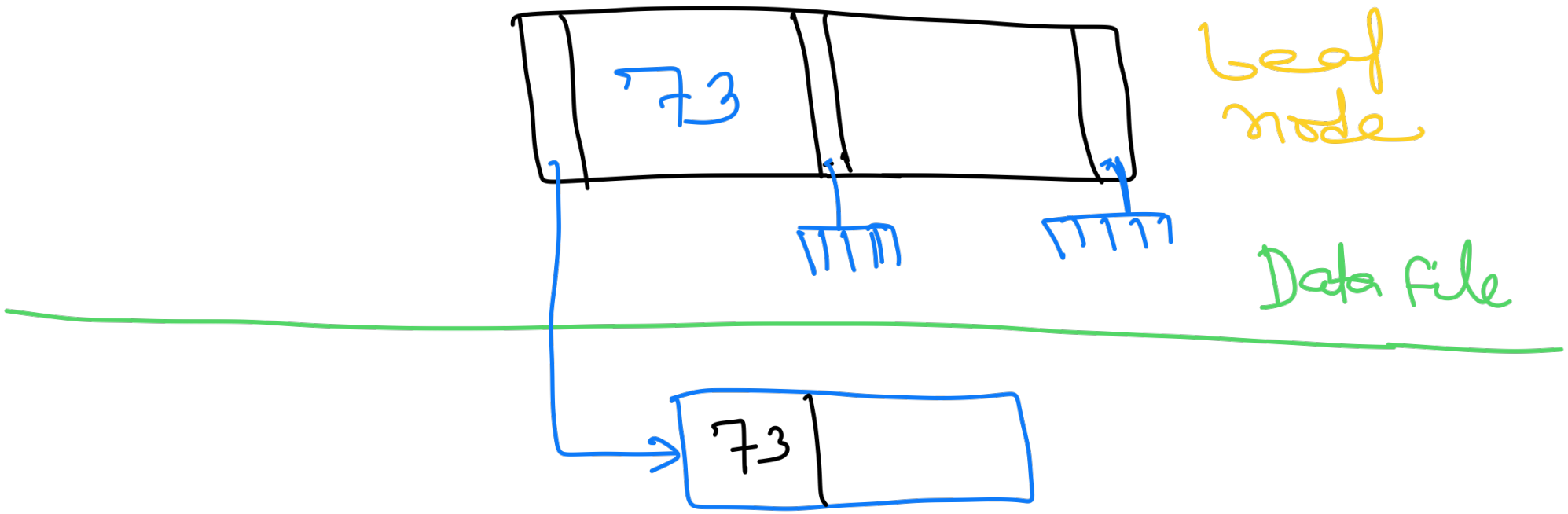
≥ 0705
< 1137

→ All index blocks have to be at least half full.

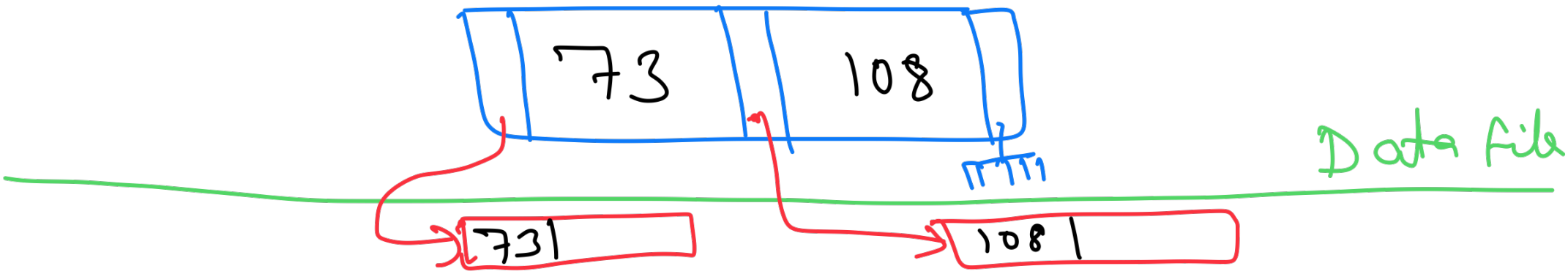
→ If K pointers are there, there should be atleast $\lceil K/2 \rceil$ pointing to valid blocks.



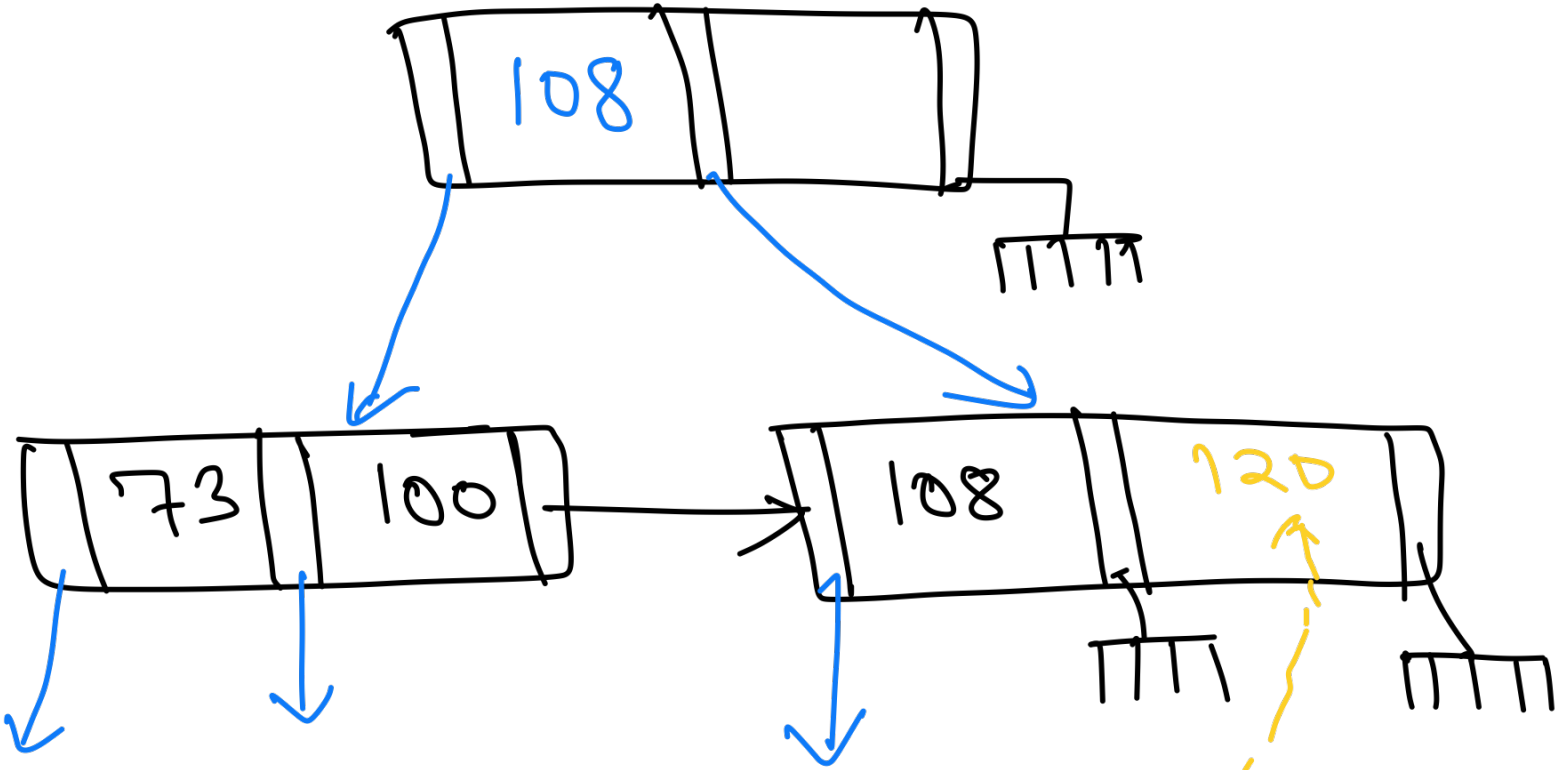
Insert 73



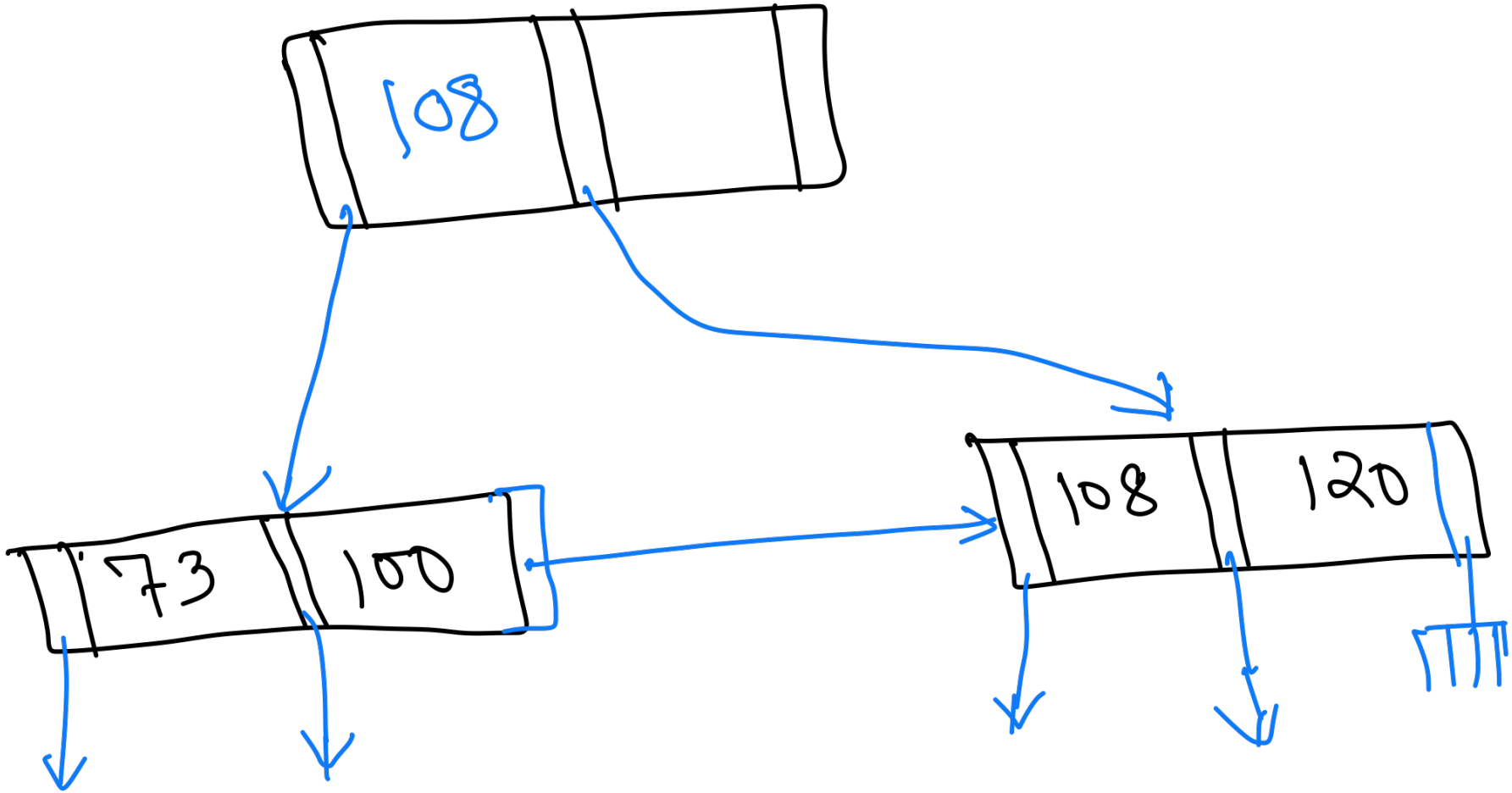
Insert 108



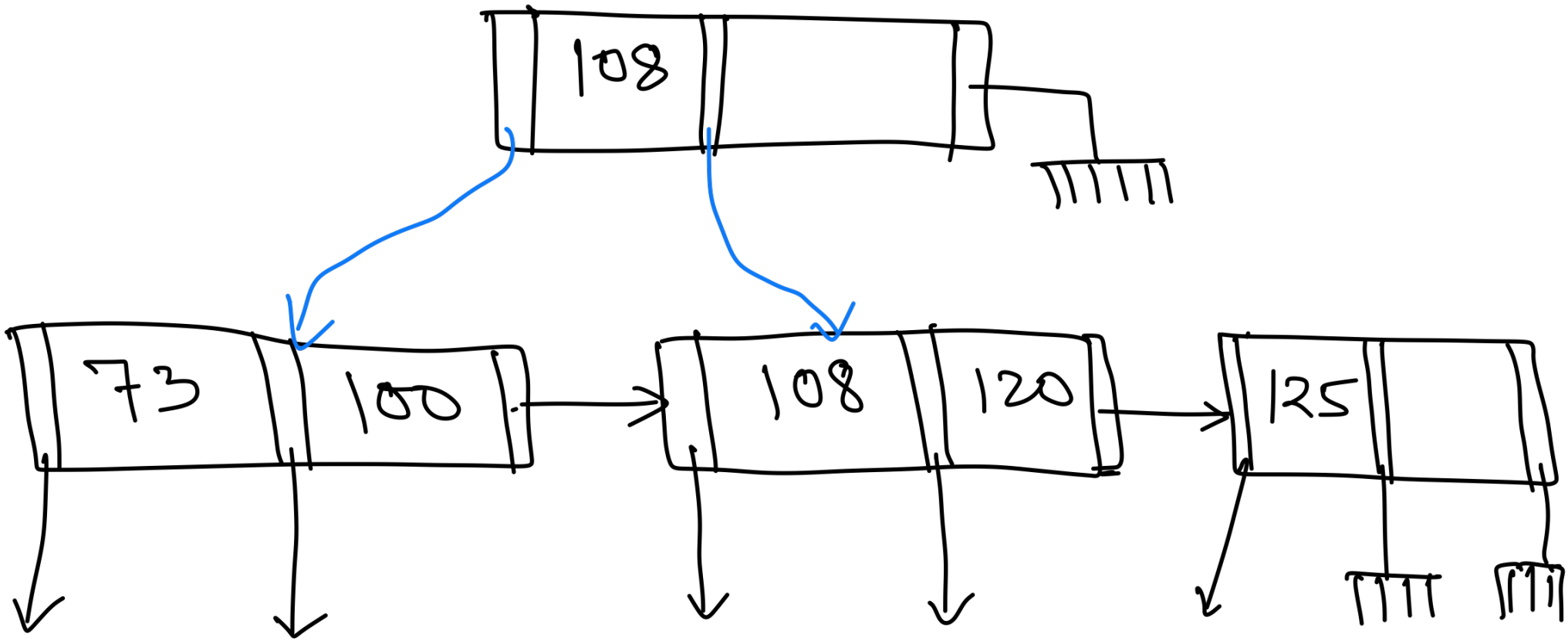
Insert 100



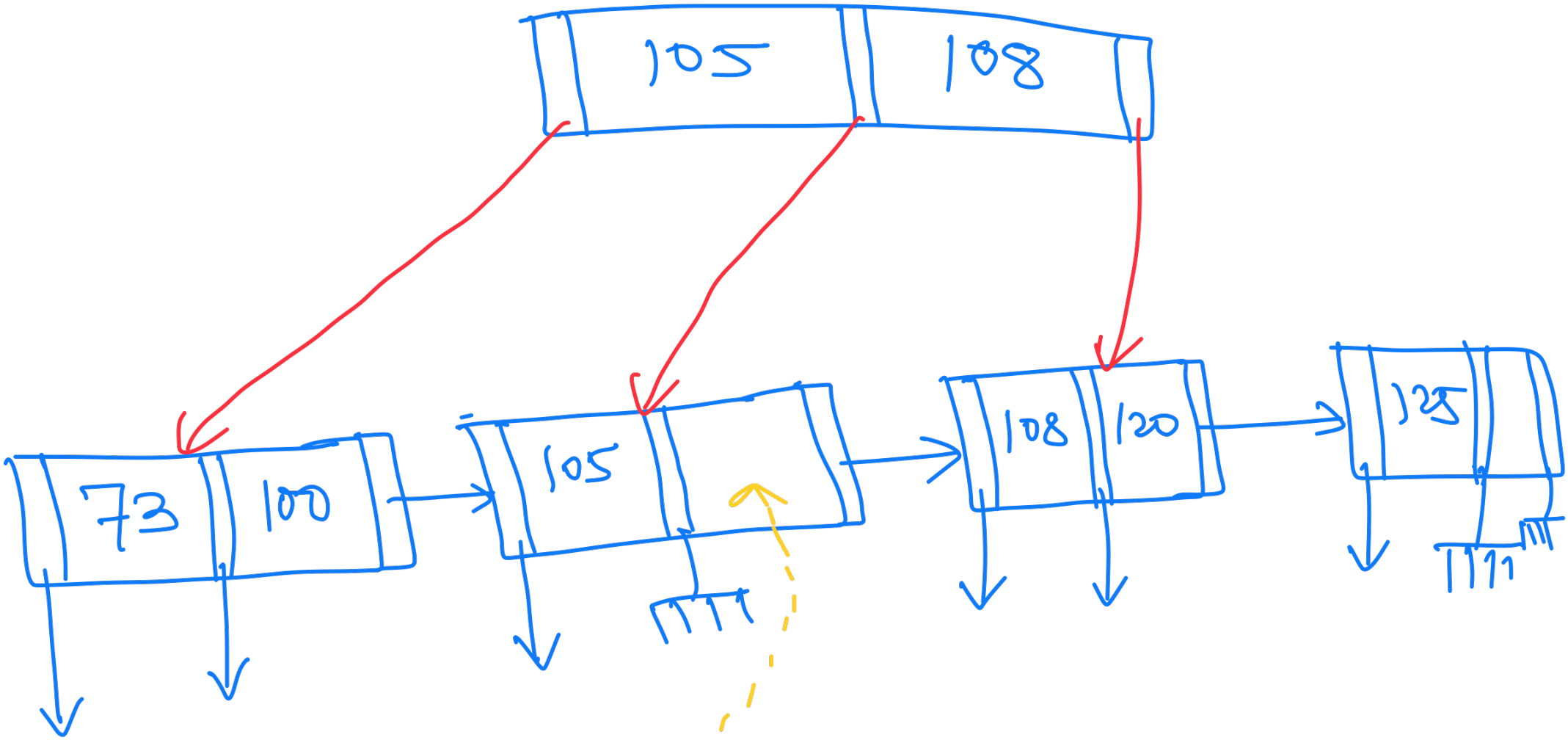
Insert 120



Insert 125



Insert 105



Insert 106

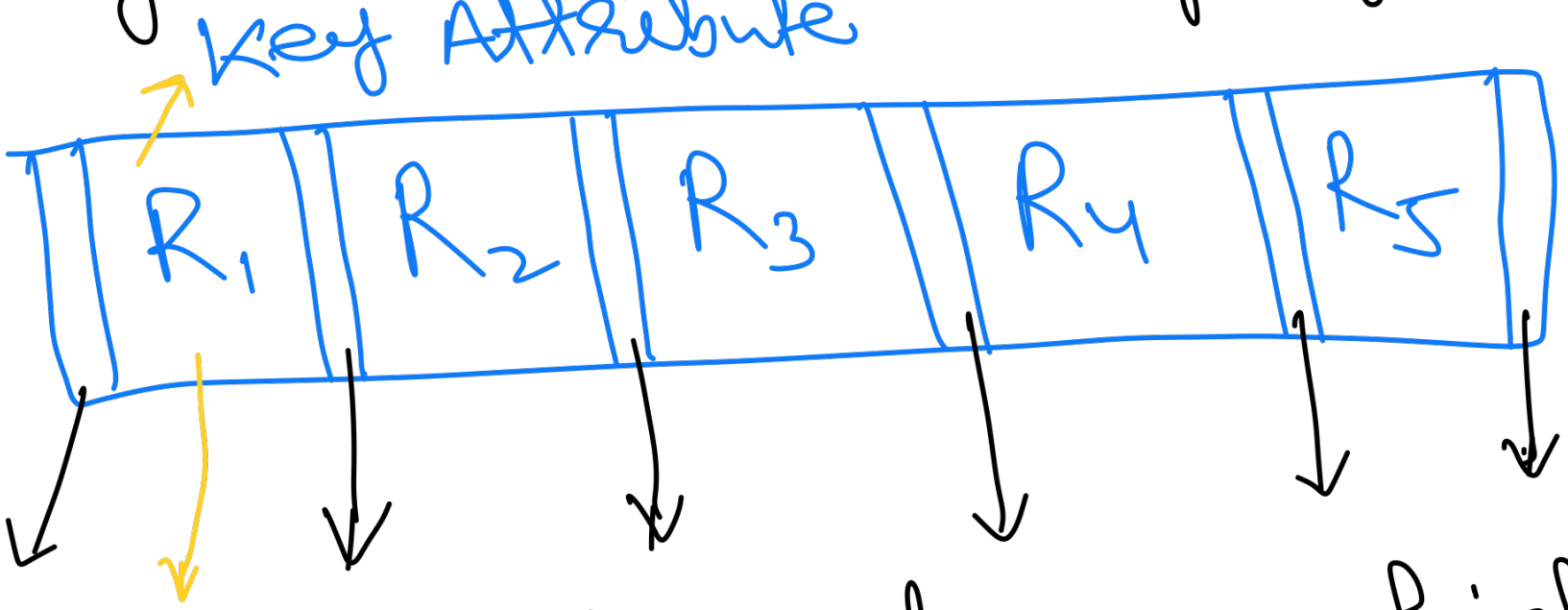
Insert 107

$$\log_3 n \leq \text{height} \leq \log_2 n$$

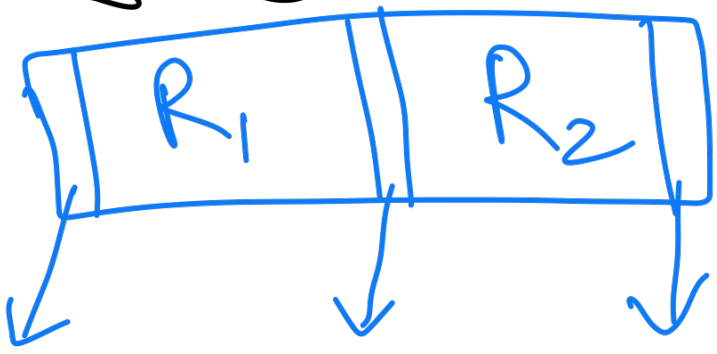
B^+- Trees

- Balanced Search Tree
- Insert, Delete, Find $O(\log n)$
- Leaf Node and Internal Node
- All leaf nodes are at the same depth on a B^+-Tree
- Each node is the size of a block

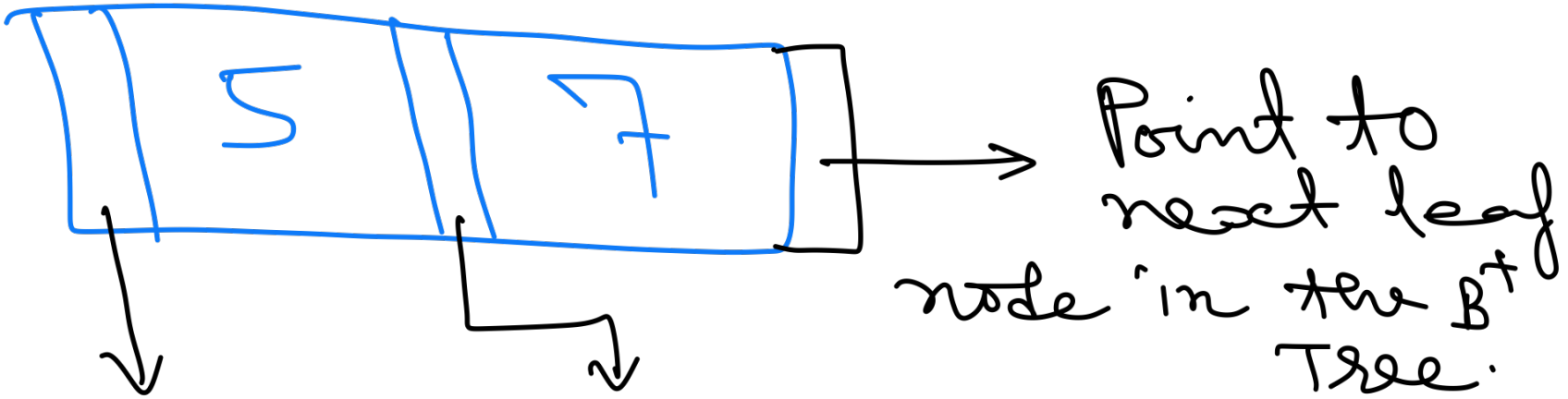
- Each node (Internal node) is always at least half full.



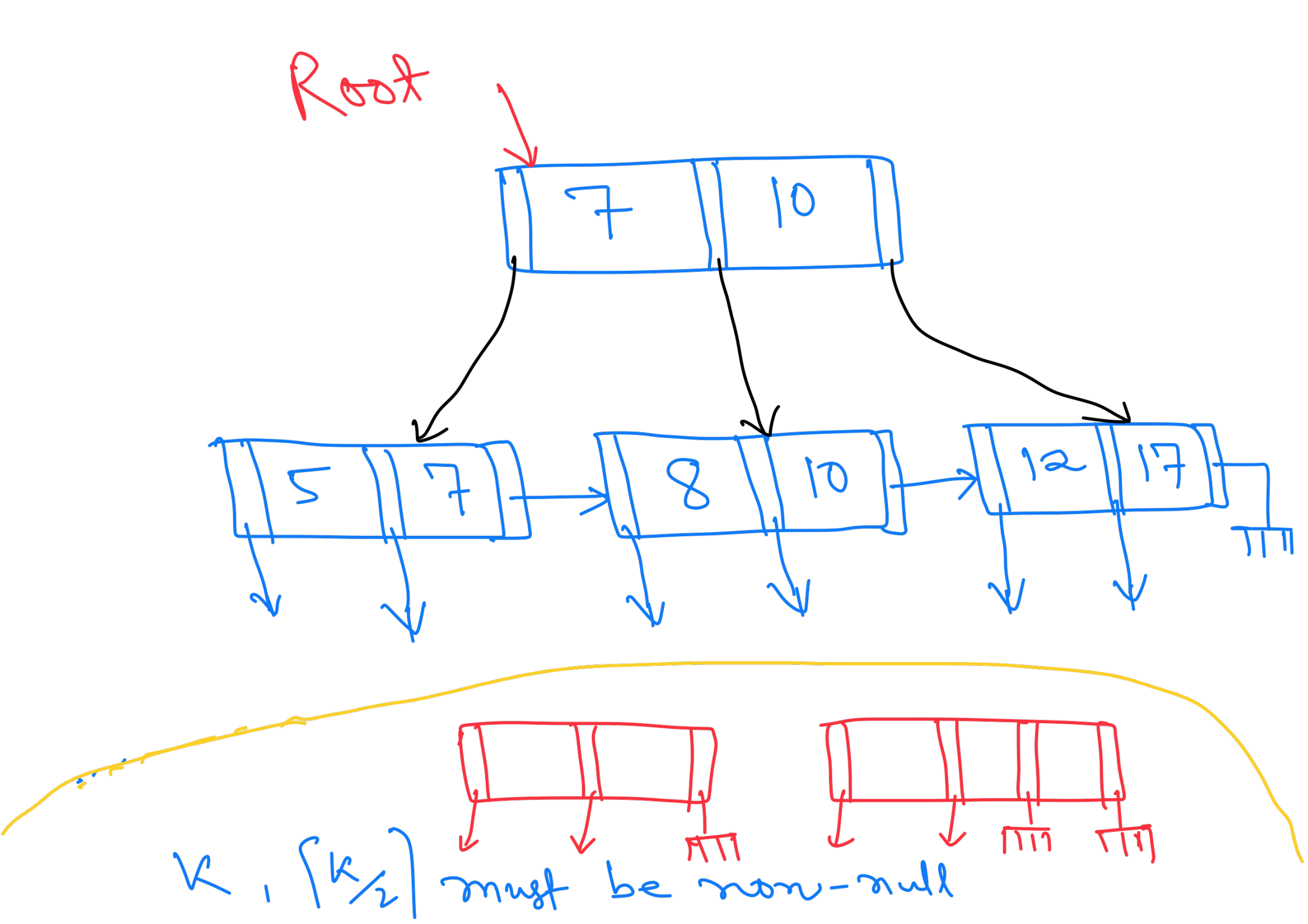
The attribute set on which indexing is done.



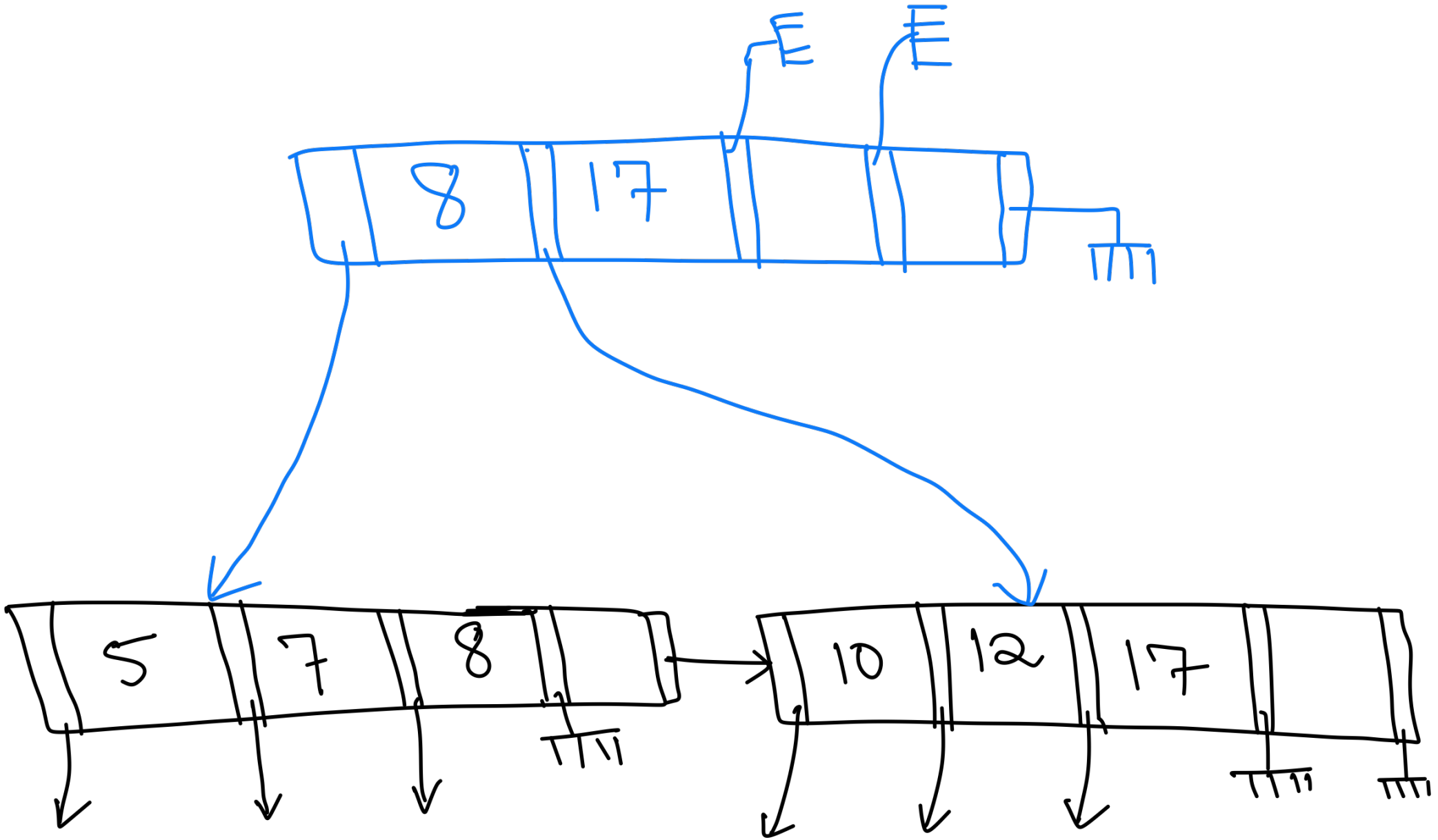
Leaf Node



Point to the block where the record corresponding to the attribute(s) value 5 exists in the main/Base file.



$K_1 \ldots \lceil K/2 \rceil$ must be non-null



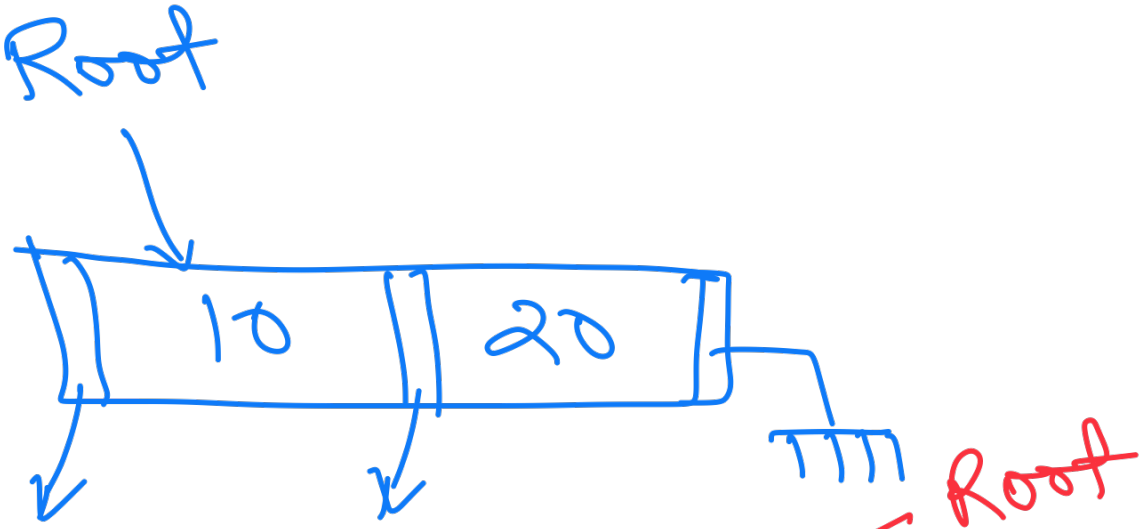
Not a valid B^+-Tree

Root

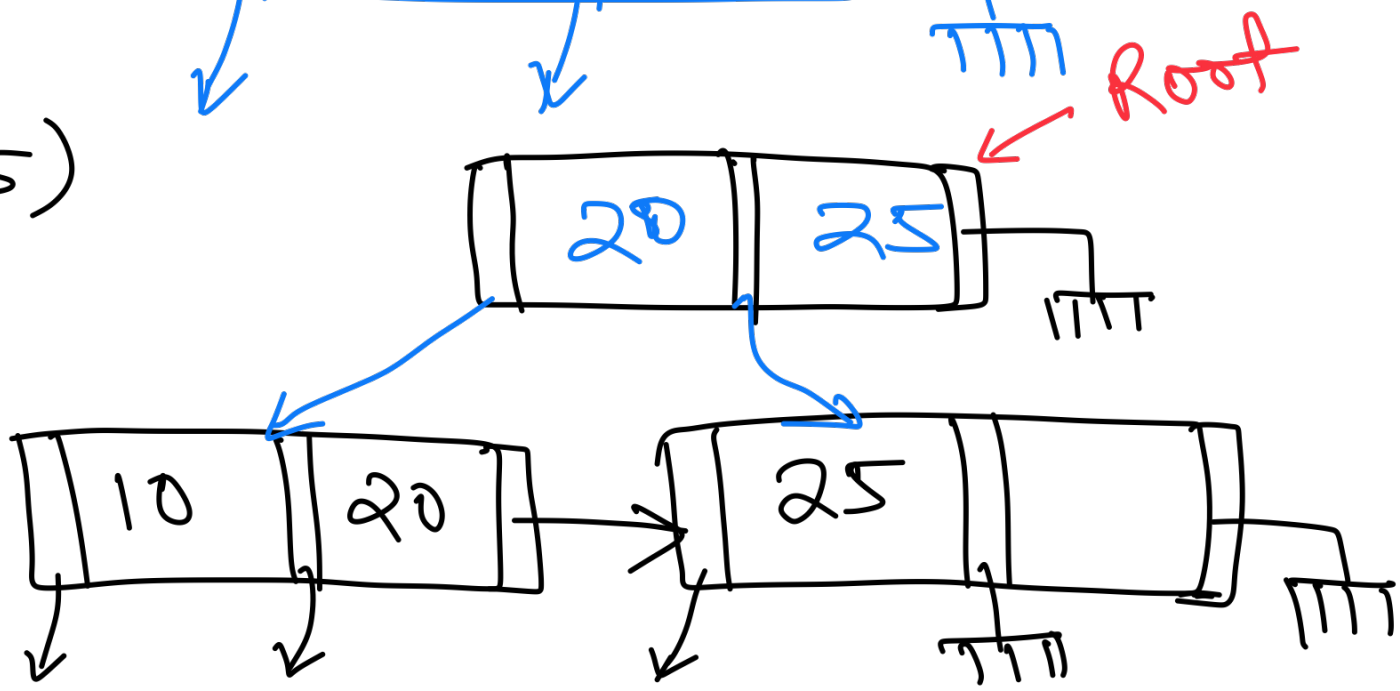
Root ⟶ Point to block where the record exists.

Insert (10):-

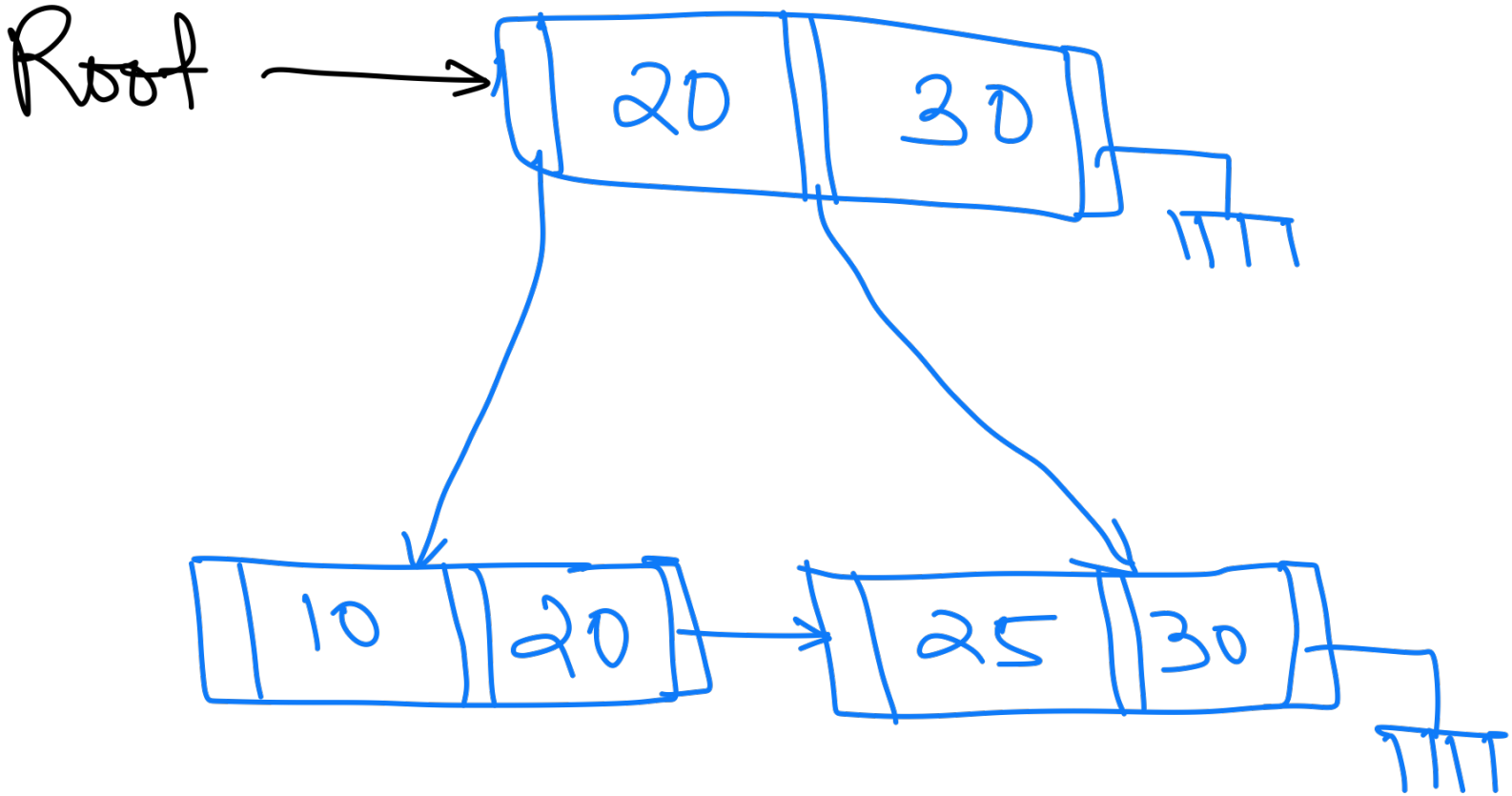
Insert (20):



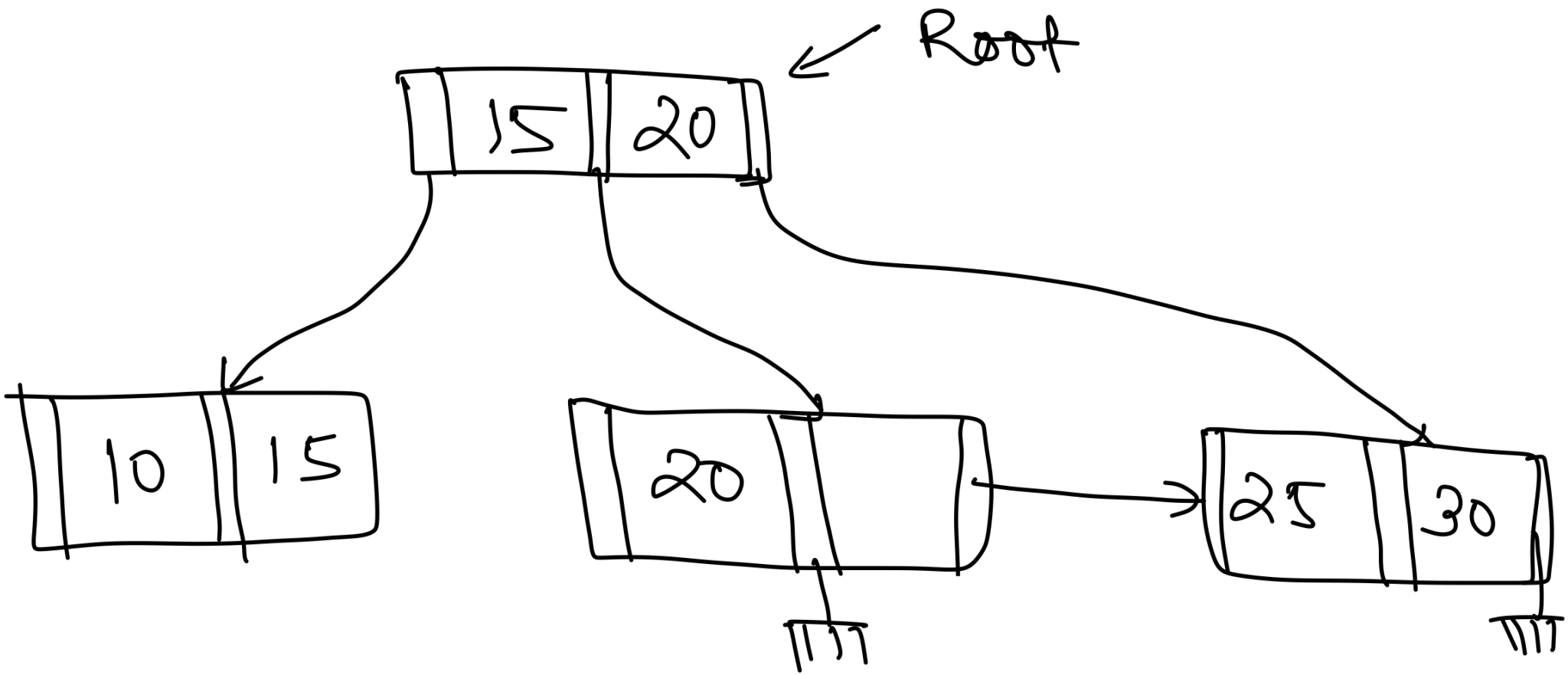
Insert (25)



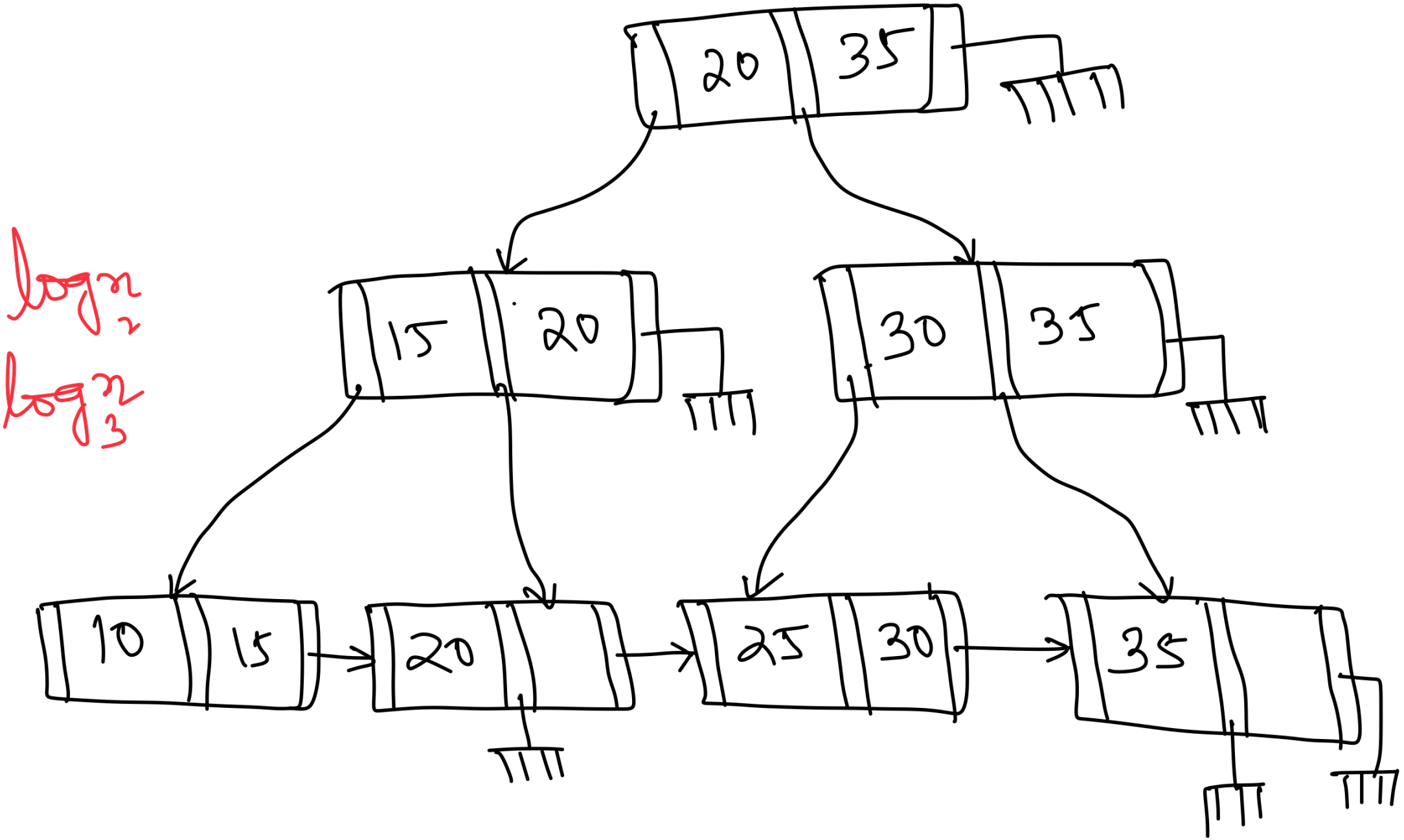
Insert (30)

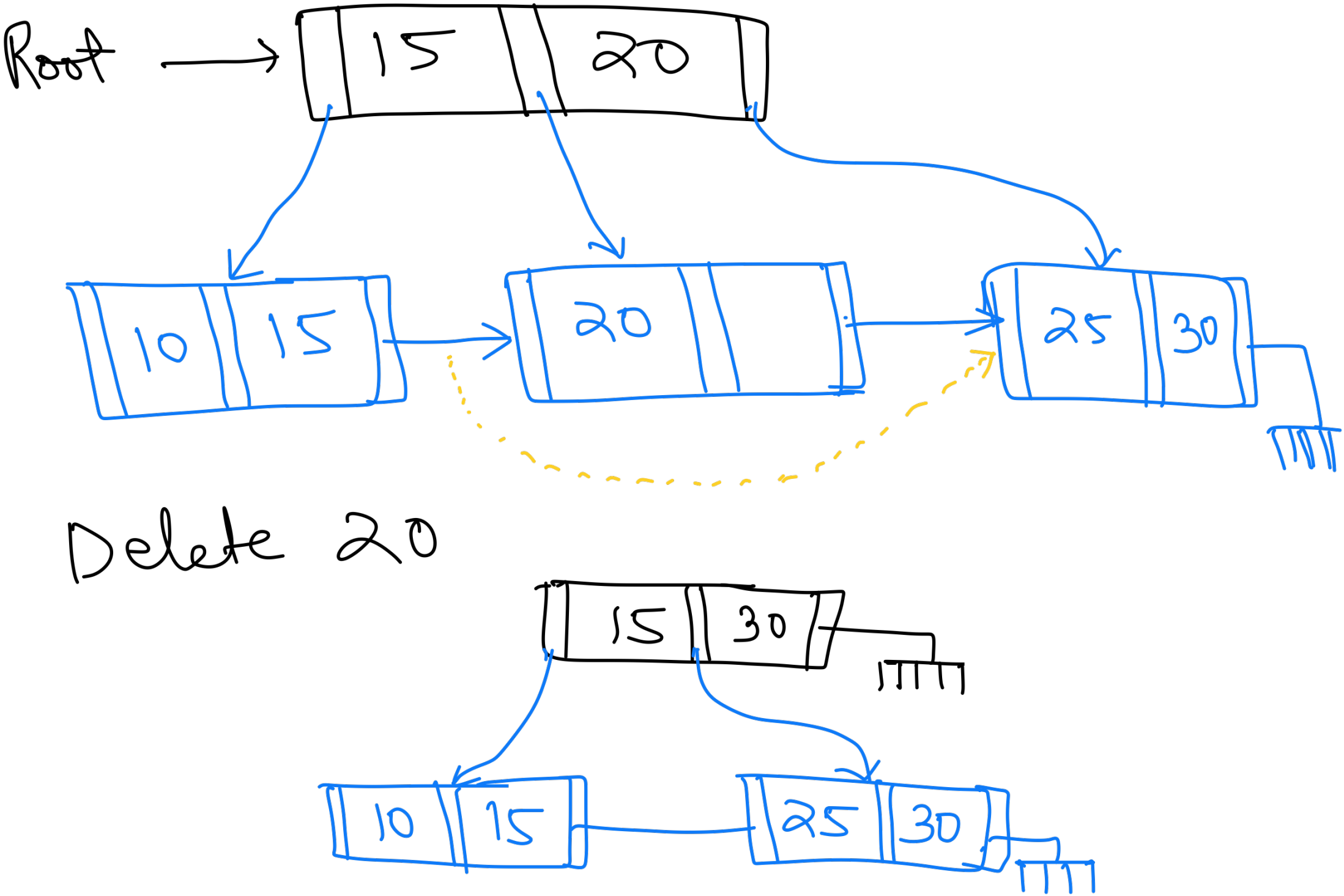


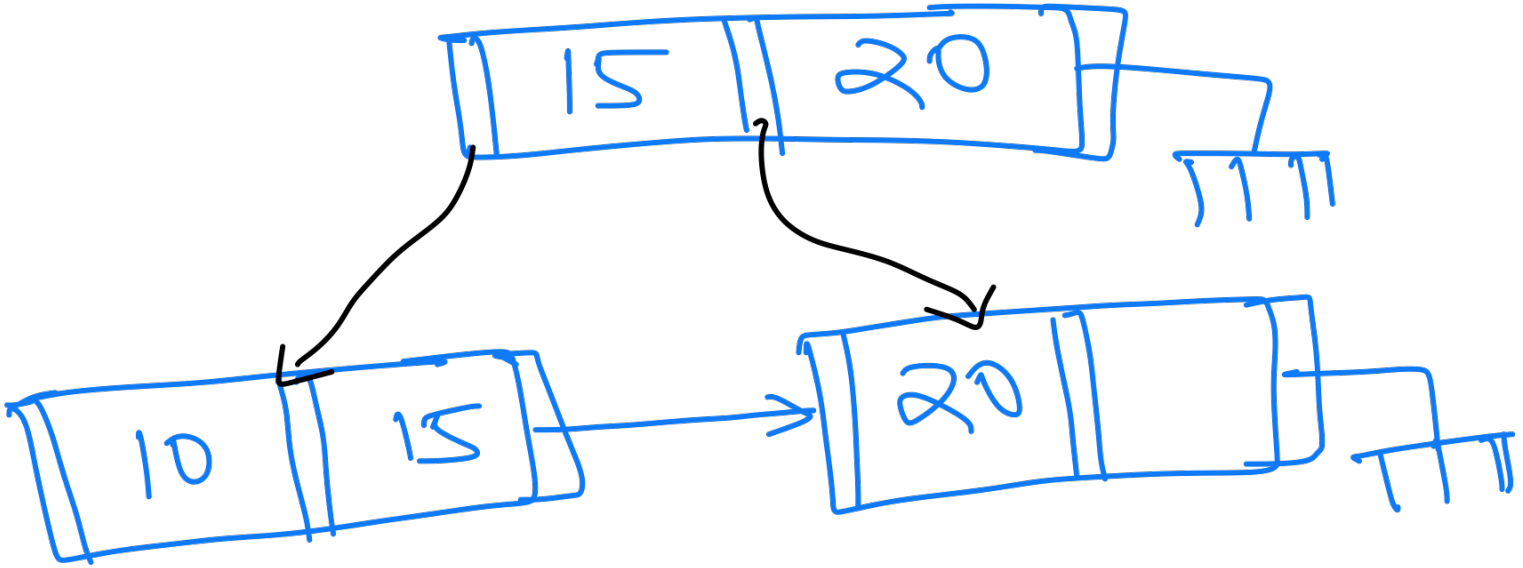
Insert (15)



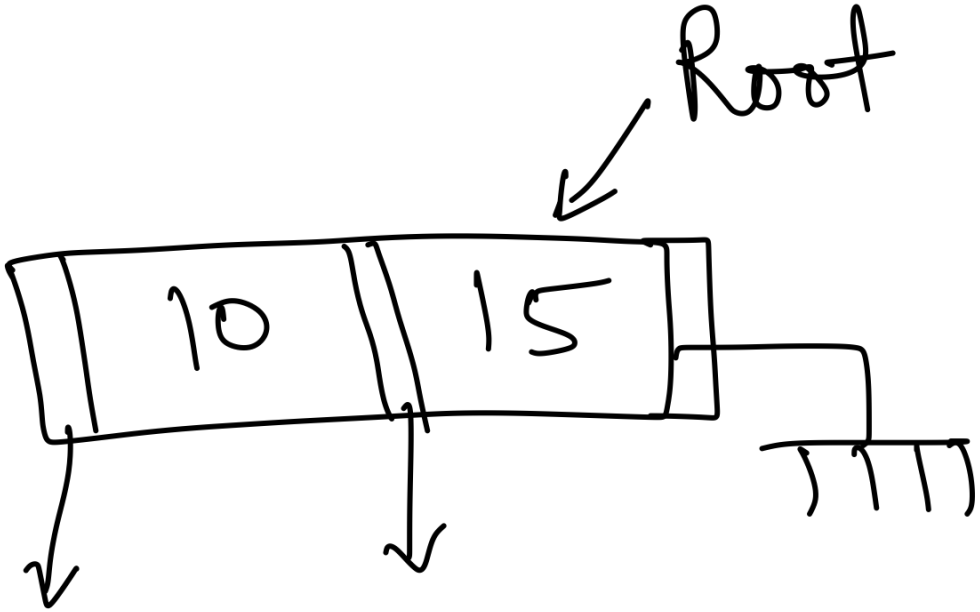
Insert 35

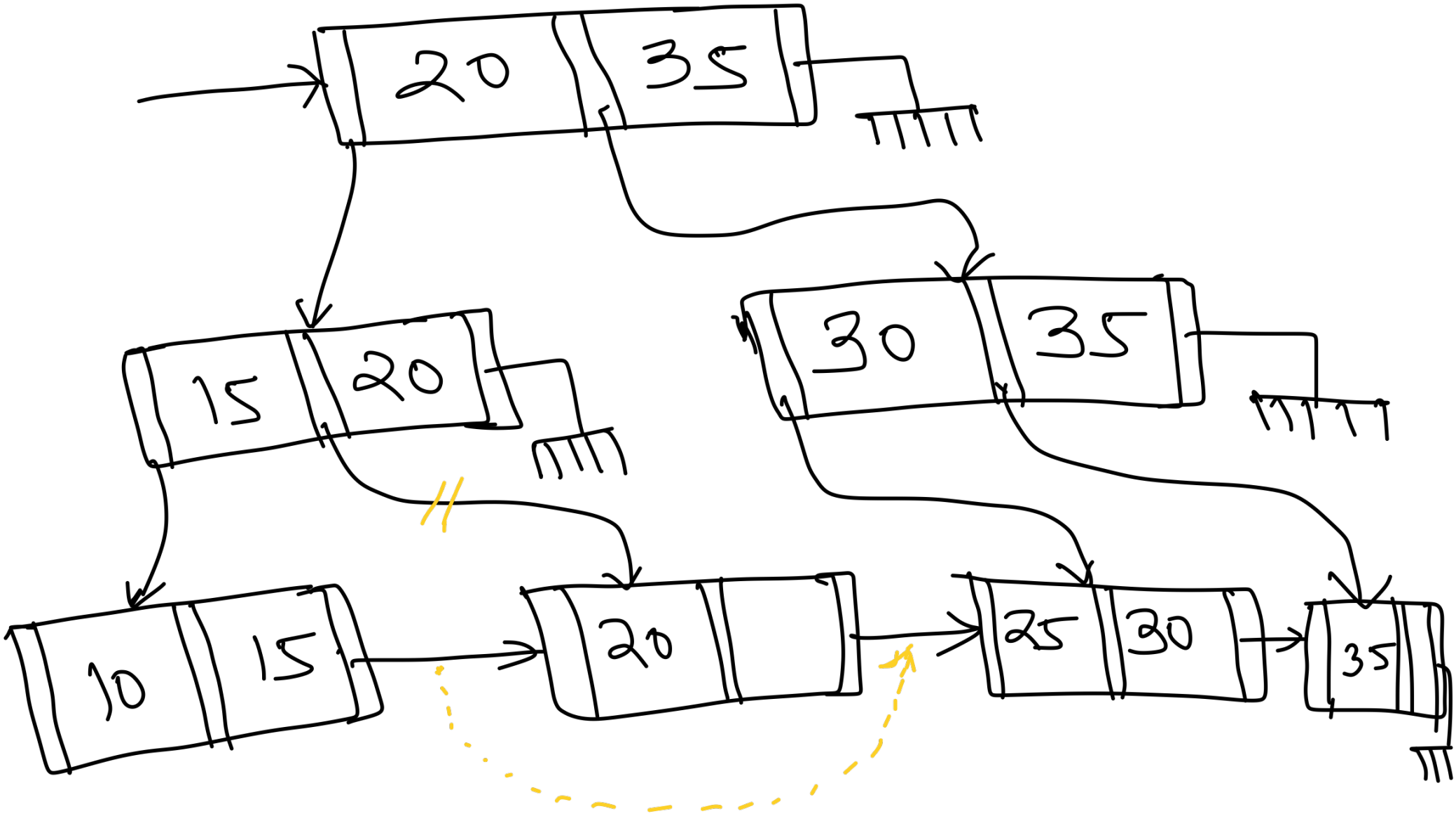

20
35
15
20
30
35
10
15
20
25
30
35
$\log_2 n$
$\log_3 n$


Root
15
20
10
15
20
25
30
Delete 20
15
30
10
15
25
30



Delete 20





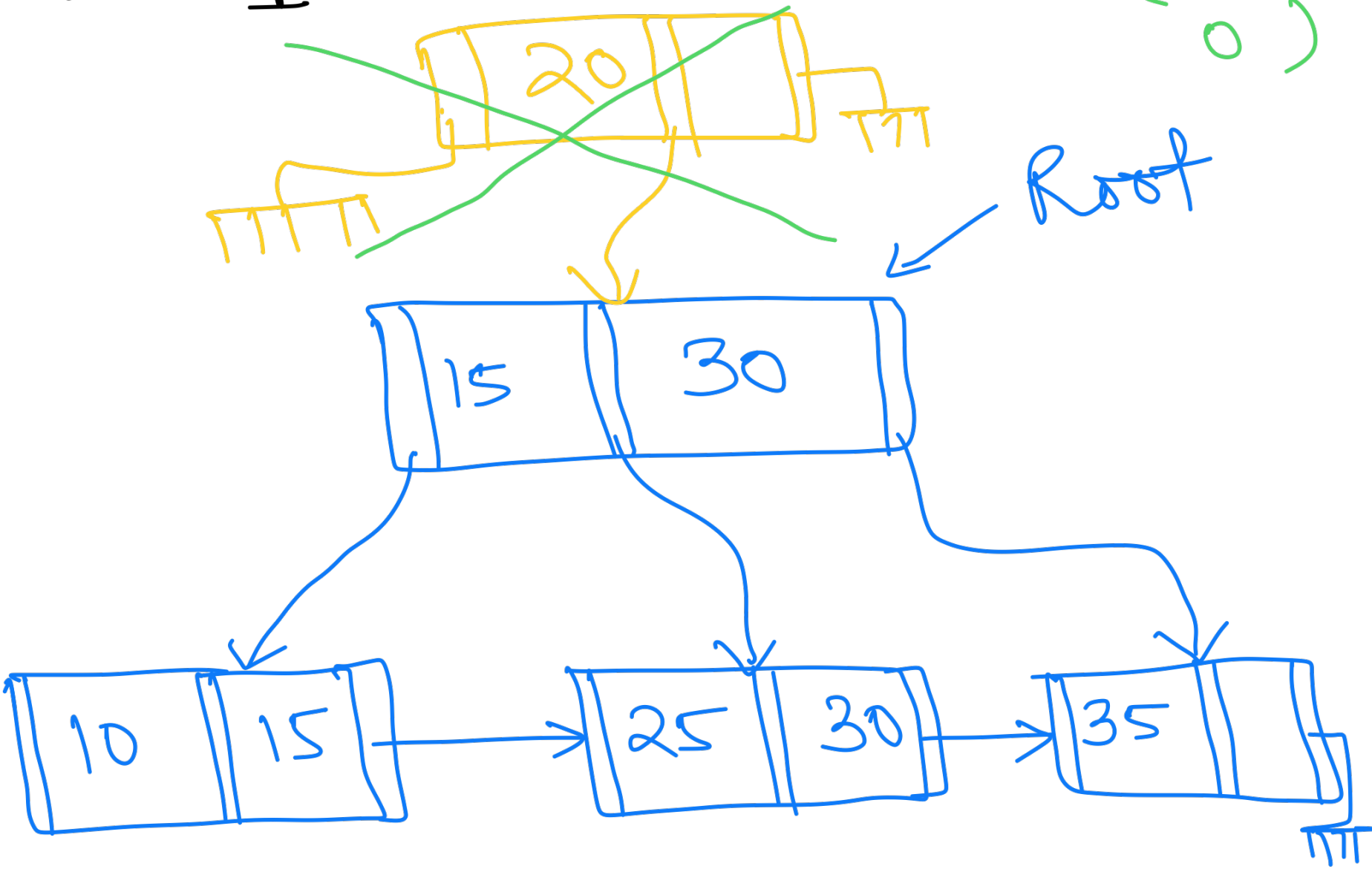
Delete 20

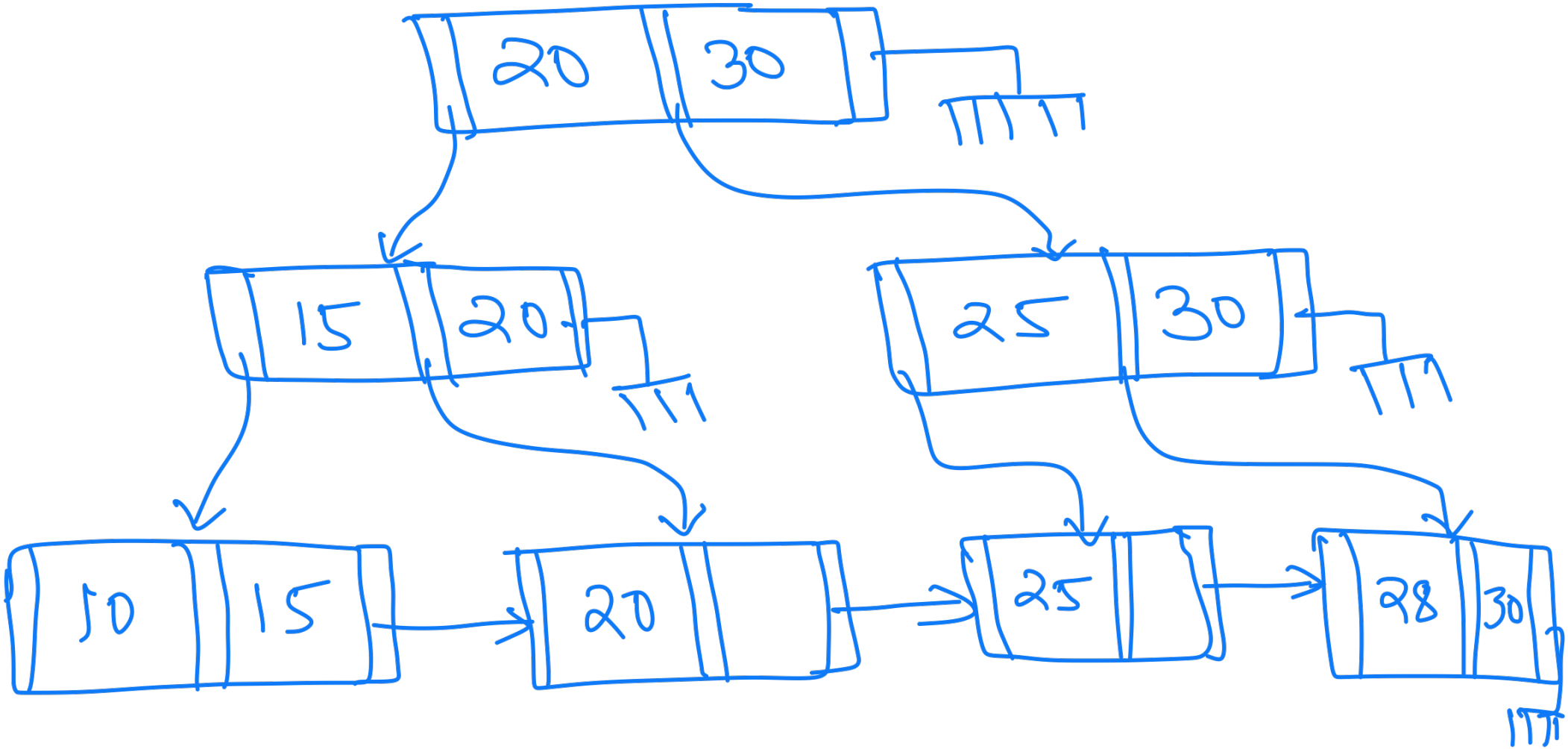
Case 1: 1 3 → Redistribute 2 2

0 3

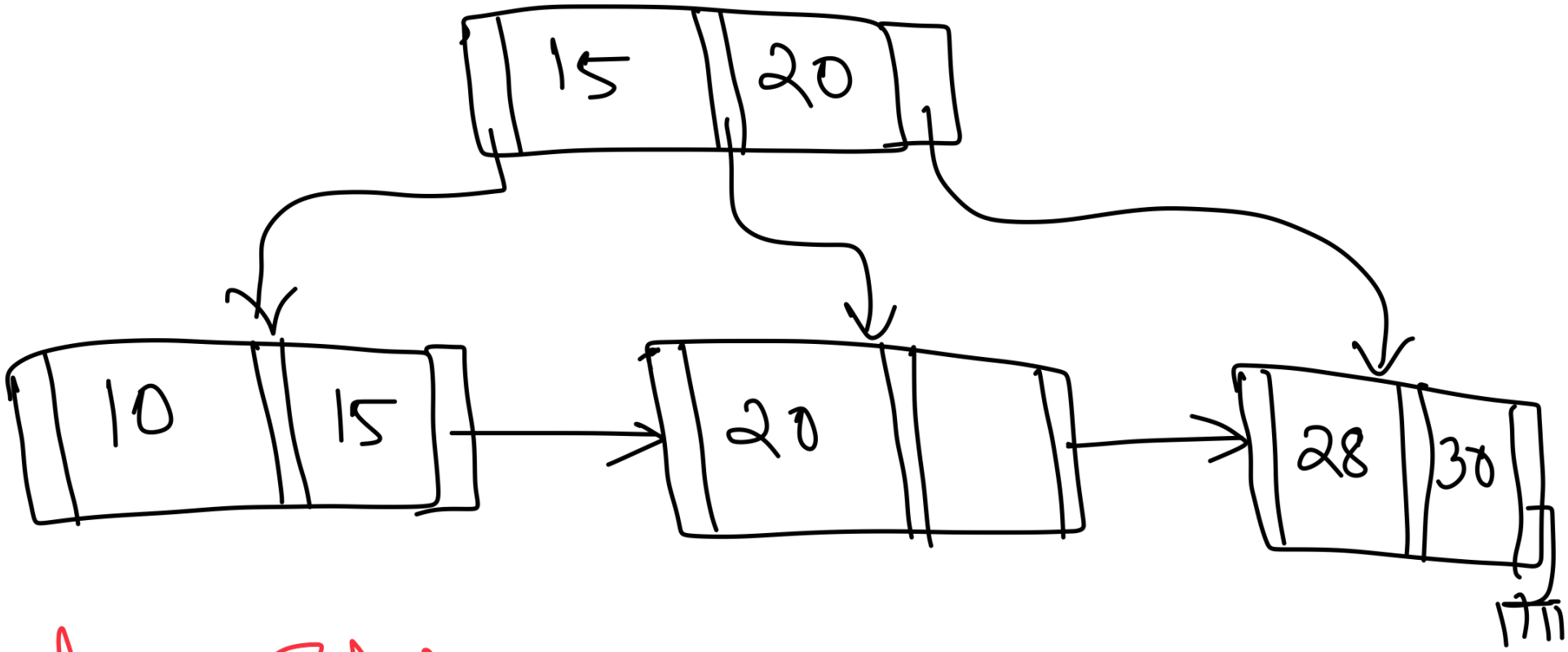
Case 2: 1 2 →

(Delete 0)





Delete 25



Delete : Steps

(i) Find the element

(ii) Delete leaf node if Required.

(iii) Delete Internal node if 11 .

Hashing : Hashed Index

$f(A) \longrightarrow$ value