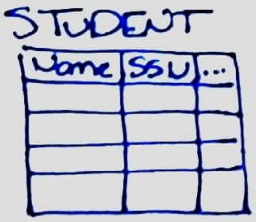
ADB Lecture 2
Indexing

- In our context, a database file stores data records of the same type (table or relation)

- An index is a file that speeds up searches (SELECT) for a Search Key (Field) on the data file



• data file

• Want to optimize searches of the form
SELECT ...
WHERE SSN = ...

→ Create An index file (index field = SSN)

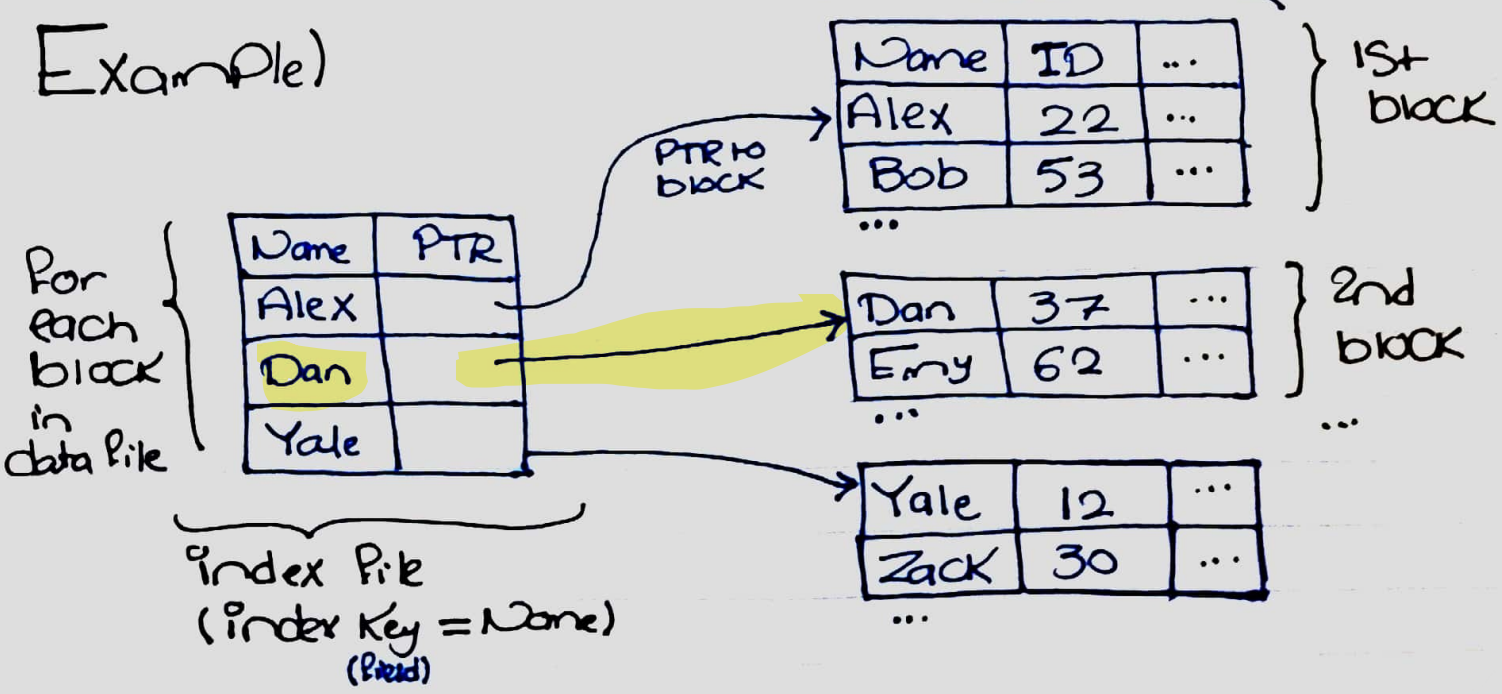
• Much faster

Search Key (don't confuse with the keys of the relation)

• The index file has two columns

Field Value (index field) — Pointer to block

Example)



- Now everytime we want to search for a record by name, we don't need to search the data file
  → Search the index file and then follow the pointer to the block
  e.g., SELECT * WHERE NAME = 'Bob'
  → Use the index to know in which block Bob will be
  - It must be the first because its clear from the index that the 2nd block starts with Dan (Dan > Bob)
  * Hence, Perform binary search on 1st block to find Bob

  (binary search on the index file.)

- Notice that this approach has assumed that the data file is sorted by the index field. (will generalize soon)

→ Ordered File with
$r = 3 \times 10^5$ records (no. of records)
$B = 4096$ bytes (block size)
$R = 100$ byte (record length)

in this case,

$$bfr = \lfloor \frac{B}{R} \rfloor = 40 \text{ records/block}$$ (B: 4096, R: 100) • blocking factor

$$b = \lceil \frac{r}{bfr} \rceil = 7500 \text{ blocks}$$ (r: 3×10^5, bfr: 40) • # blocks

Hence, the no. of I/O operations is

$$\#\text{Io}_{(\text{disk})} = \lceil \log_2 b \rceil = 13$$ (b: 7500)

- Let's see how would indexing help
  → Suppose the file is ordered by a key of 9 bytes and that block pointer is 5 bytes

- Then each row in index file has R_{ind} = 9+6 bytes
  → and it has only 7500 rows (one for each block) $b = r_{ind}$

- How many blocks will we need to store the index file?

$$bfr_{ind} = \left\lfloor \frac{B}{R_{ind}} \right\rfloor = 273 \text{ index records/block}$$ (B = 4096, R_{ind} = 15)

$$b_{index} = \left\lceil \frac{r_{ind}}{bfr_{ind}} \right\rceil = 28 \text{ blocks}$$ (r_{ind} = 7500, bfr_{ind} = 273)

- # index blocks

$$\#IO = \lceil \log_2 b_{ind} \rceil + 1 = 6$$

- $\lceil \log_2 b_{ind} \rceil$: to find the index block
- 1: to read the block index points to (data block)

Mem.

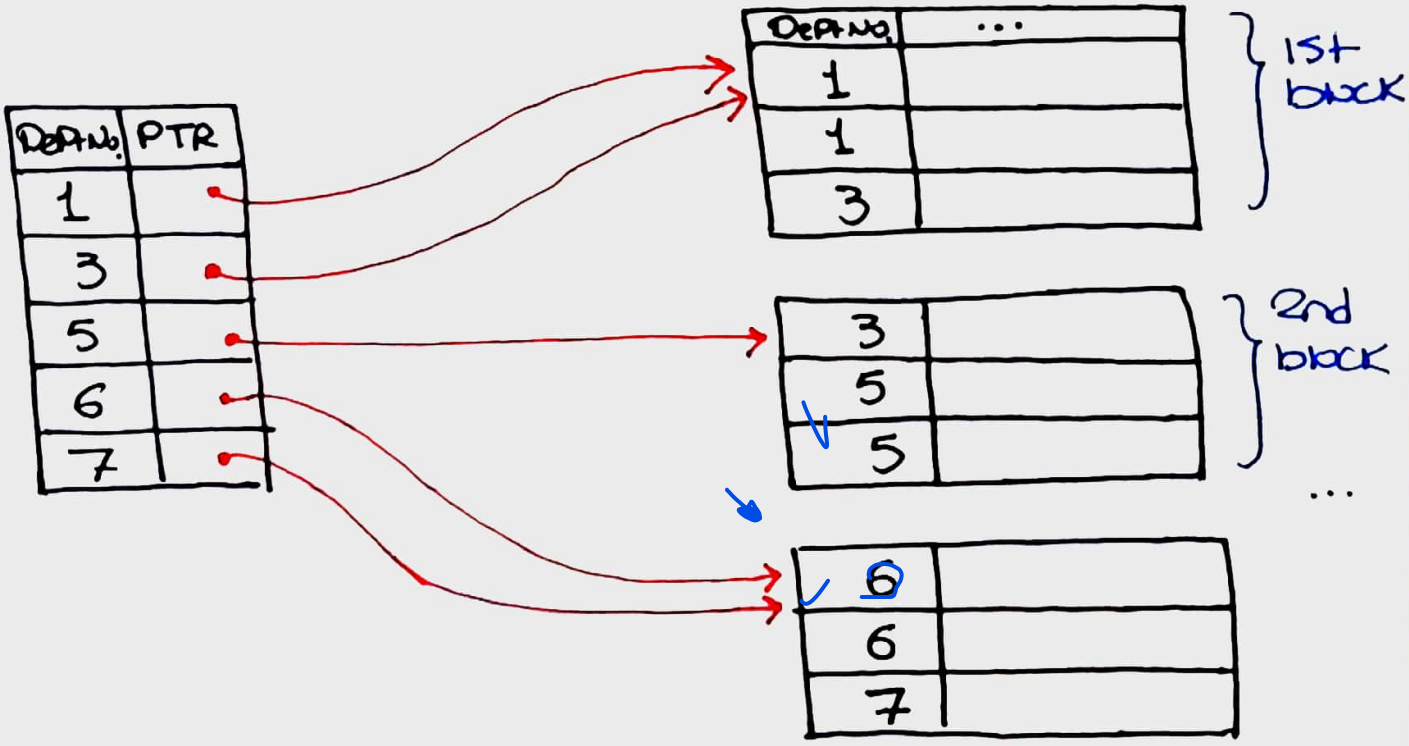
* There are 4 types of indexes
  - Primary index
  - Clustering index
  - Secondary index (Key)
  - Secondary index (NonKey)

→ What we've seen so far is the Primary index
  - Requires that data is ordered by a Key field (unique or Primary Key)
    ↳ Candidate Key ↳ unique by definition

  - In this case we create the index using the first element of each block as the value and the block's address (1st record addr.) as pointer "anchor"

→ Notice that the index is SPARSE (we don't have a pointer for each record in the data file; rather, for each block)
  - which is why the index is smaller (more efficient)

- What if the data is ordered by a field that isn't unique? (non-key field)
  - → an index is still possible
  - → For each unique value of the field, make a record in the index.



- For each unique value, the block pointer points to the block where that value has 1st occured
  - → Select * WHERE DEPTNO. = 3
  - → Select

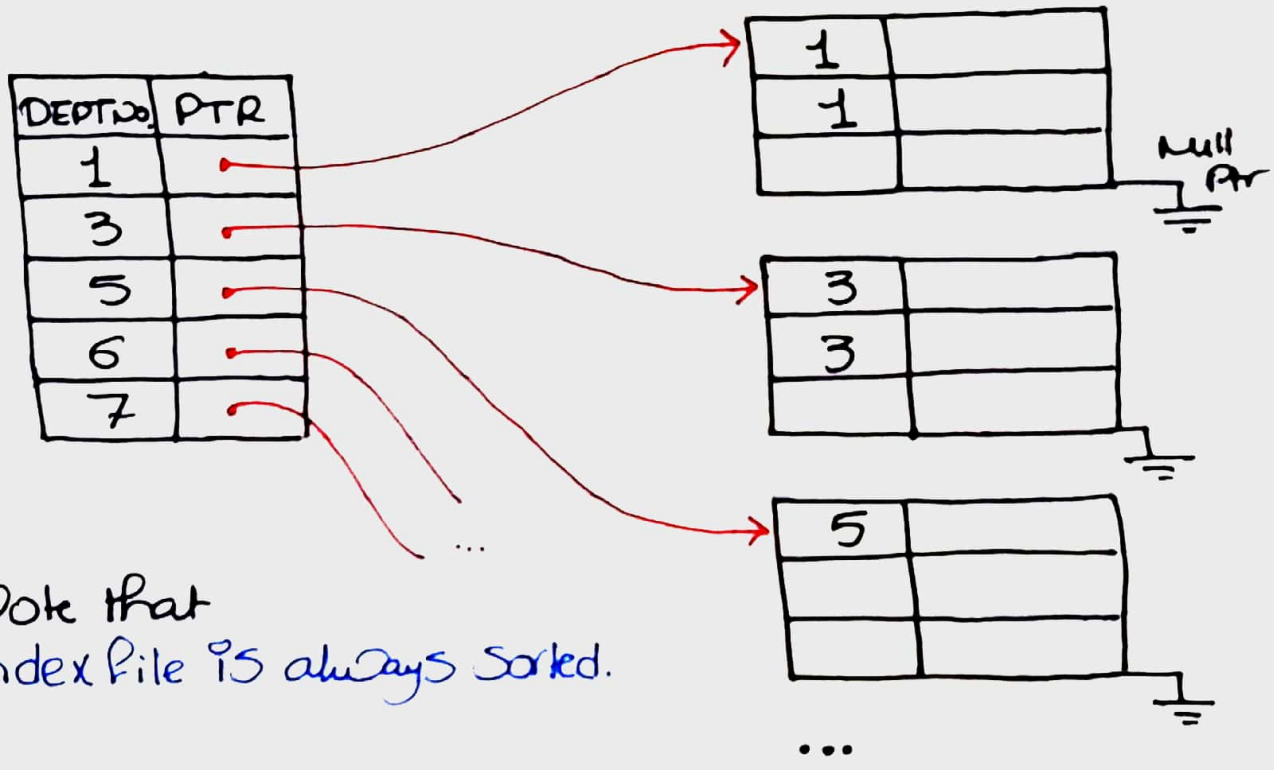
1. Search in index for 3 (binary search)
2. Read the 1st block and find 3 in it
3. Since last record is 3 you have to read next block (might be a continuation)

- → Select * WHERE DEPT NO. = 5

Something but
- → 2nd block is returned
- → No need to read next block

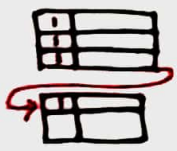
- Notice that insertion/deletion in this case isn't so bad (many pointers to move)

Alternative Design (uses block anchors)



- Note that index file is always sorted.

* In the data file, each group sharing the same value of index (clustering) field should have its own block
  - → insert into free space in block (assuming value exists)
    # That's it
    # NO PTR fix
  - → If it doesn't fit
    - make a new block
    - chain it with the original one (let the original one point to it) e.g.



* Clearly in this approach
  - rind = # unique values (# index records)
  - Sparse index

Example

- Ordered File with
  $r = 3 \times 10^5$ records
  $B = 4096$ Bytes
  $R = 100$ Bytes
  → $bfr = 40$ records/block, $b = 7500$ blocks, $\#IO = 13$ (last Problem)

→ The ordering attribute is ZipCode

→ Although File has 3×10^5 records, it holds only 1000 unique ZipCodes • $r_{ind} = 1000$
- i.e., on avg. 300 records have the same ZipCode

* What is #IO for a clustered index.

$$bfr_{ind} = \left\lfloor \frac{B}{R_{ind}} \right\rfloor = 273 \text{ index rec./block}$$

(B: 4096; R_{ind}: assume 15 (like last Prob.))

$$b_{ind} = \left\lceil \frac{r_{ind}}{bfr_{ind}} \right\rceil = 4 \text{ blocks}$$

(r_{ind}: 1000; bfr_{ind}: 273)

$$\#IO = \lceil \log_2 b_{ind} \rceil + 1 = 3$$

(b_{ind}: 3)

- This will help us only read the 1st block but
  → on average 300 records have same zip and each block can hold 40 records. So to read them all we'll IO in few more blocks.

* Note that any File can only have one Primary Index or Clustering index (not both)

→ It's either ordered by a unique or non-unique field

- What if we want more?
- What if the file isn't ordered by the search key for which we want queries to be optimized?

Key

* In this case we can use a Secondary Index

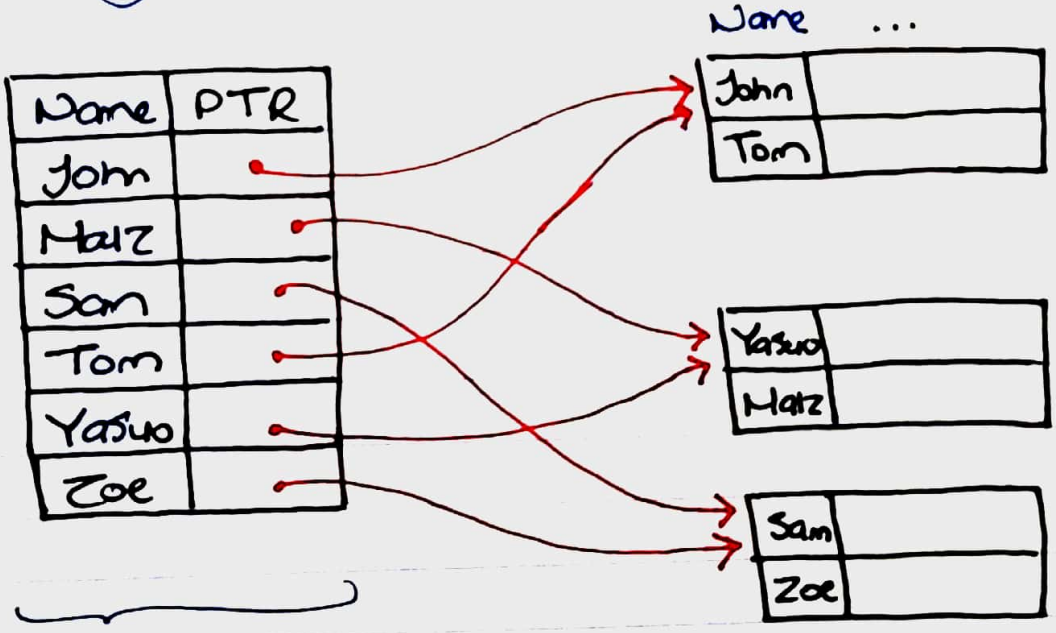
→ In both cases, data file doesn't have to be ordered by index field

(Key) type
- index field is unique

(Non-key) type
- index field isn't unique

(i.e., can be created as many times we wish besides the Primary/Clustering index)

- Secondary Index (Key)



The index is still sorted
→ has a block pointer for each record on data file (dense)
- Binary search on it then I/O on the block it points to.

Example

Solved earlier:
- bfr = 40 records/block
- b = 7500 blocks
- #IOs (sorted) = 13 (else 7500/2)

← $r = 3 \times 10^5$ records, $B = 4096$ Bytes, $R = 100$ Bytes

- $R_{ind} = 15$
- $bfr_{ind} = \lfloor \frac{B}{R_{ind}} \rfloor = 273$ index record/block (B ← 4096, R_{ind} ← 15)

Solve for #IOs:

$r_{ind} = r = 3 \times 10^5$ index records (dense)

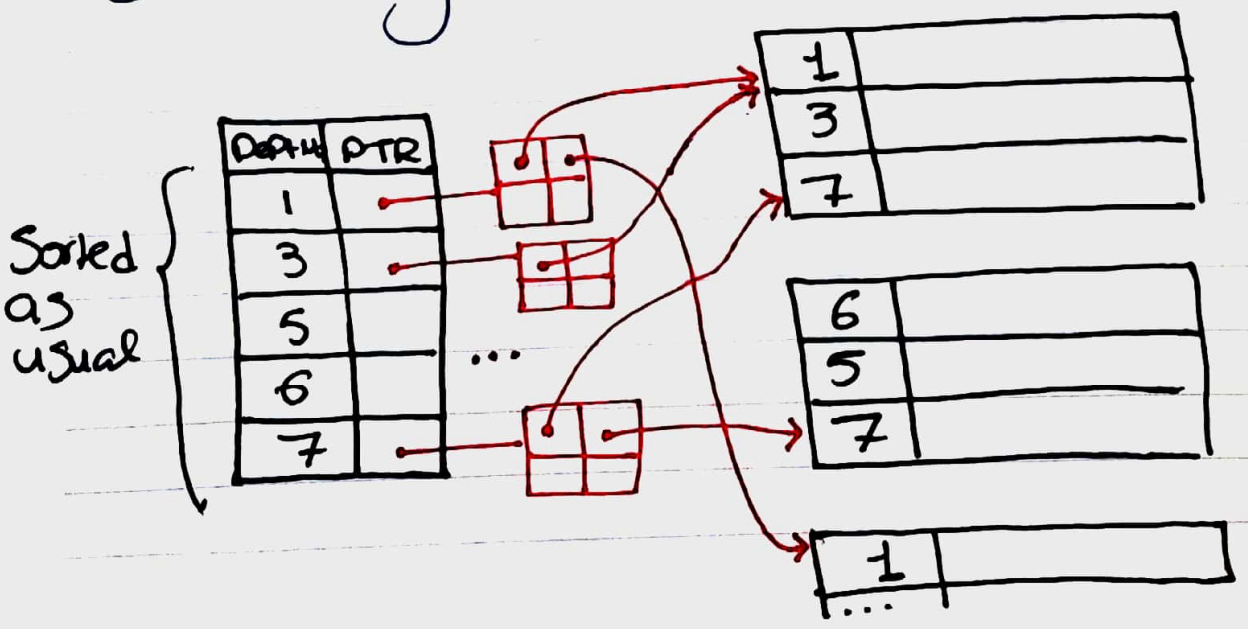
$b_{ind} = \lceil \frac{r_{ind}}{bfr_{ind}} \rceil = \log g$ block (r_{ind} ← 3×10^5, bfr_{ind} ← 273)

$\#IOs = \lceil \log_2 \log g \rceil + 1 = 12$

- Savings due to sort → smaller record

- What if we want to optimize search queries where
  - data file not sorted w.r.t search key.
  - key is not unique

- Secondary Index (Non-Key)

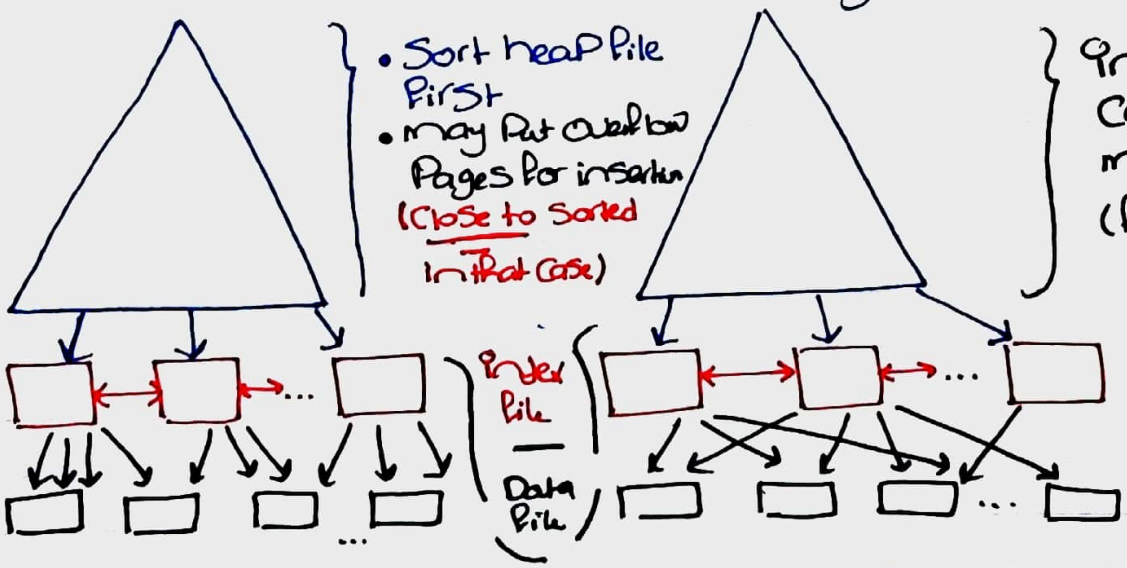


- Each unique index points to a block of pointers
  → The block has pointers to all data blocks where the index value occurs

- Summary

		Kind	Sparsity	Block Anchor (→ each block = 1 Ptr)
Need Sort	Primary (Key)	b	Sparse	✓
	Clustering (Non-Key)	#uniques	Sparse	✓ or ✗
Don't need Sort	Secondary (Key)	r	Dense	✗
	Secondary (Non-Key)	#uniques (other variant → r)	Sparse (other var → Dense)	

- Clustered VS. Unclustered Index (Secondary) } Cost of retrieving records differ (as was seen)



- At most one clustered index for any file (one search key)

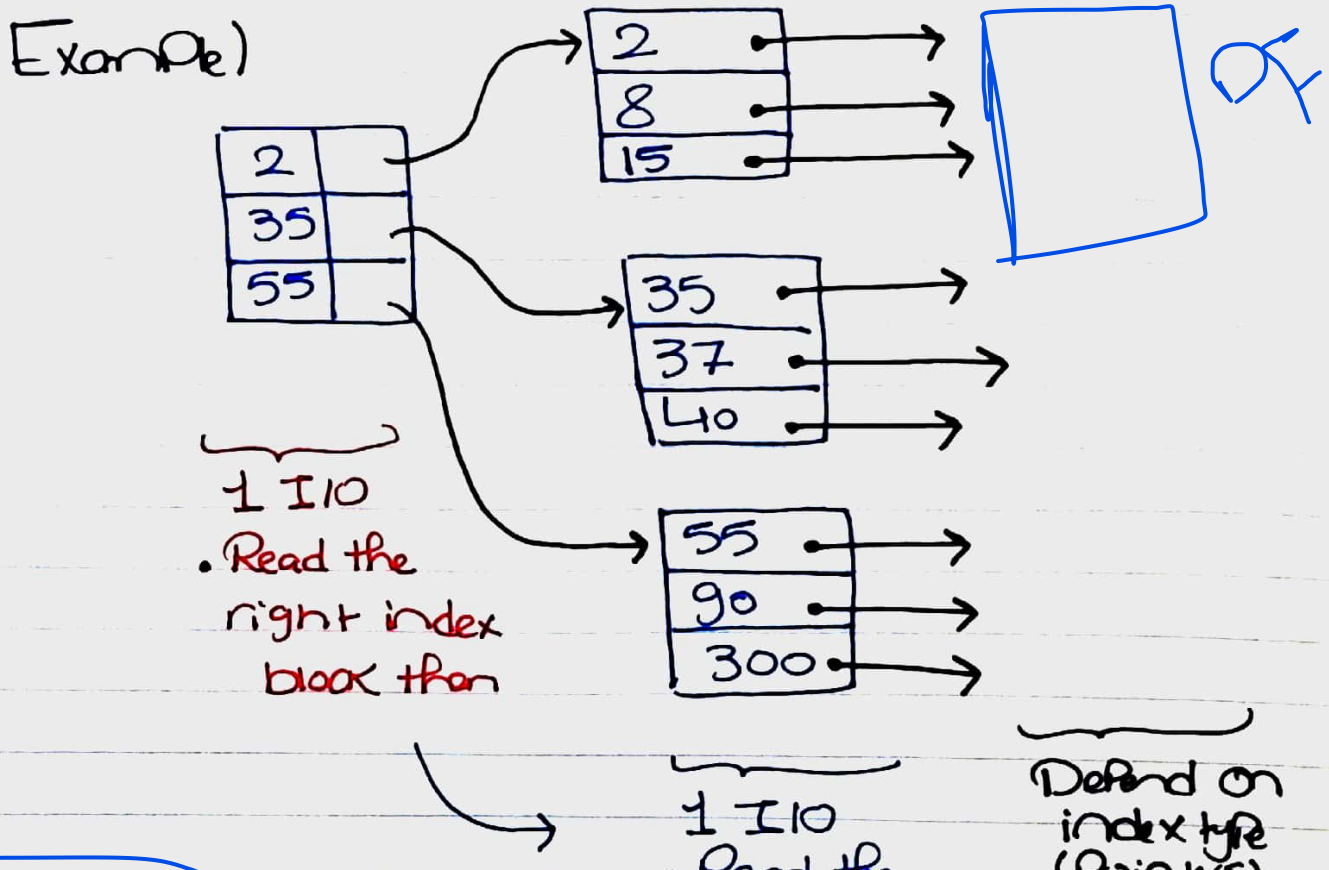
• Multi-level Index

→ Single-level index is an ordered file

→ Can hence always create a Primary index on the index itself
- Upper-level would be called second-level index

* Don't have to stop at 2nd level
  → Can keep going until top most level is one block
  - Search becomes O(# levels) rather than blocks

→ • Type of 1st level doesn't matter (only should be more than one block)



#ISAM
(indexed Seq. access method)

Problems with Index

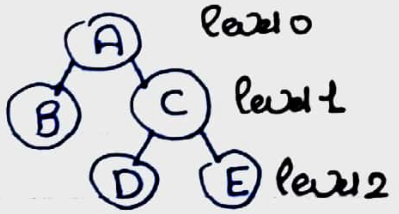
→ Deletion
→ use markers (soft delete)

→ Insertion
→ Insert in right position in index
→ May need to move records in data file, may affect anchor records (intuitively)
• Linked list or overflow records may help?
→ Need a more dynamic solution.
• A dynamic index isn't statically allocated and can grow/shrink as needed due to insertions/dels

Can either be tree based or hash based.

• Tree-based Dynamic Index (Multi-level)

→ Example of a tree



not balanced
→ left and right subtrees not always of same height.

~~• when a tree satisfies that for any node~~

* Search Tree
→ Each node consists of P-1 keys
→ To the left and to the right of each key there's a pointer to a node in the lower level (or nullptr).
• Hence, P pointers for each node, each pointing to some sub-tree(node).

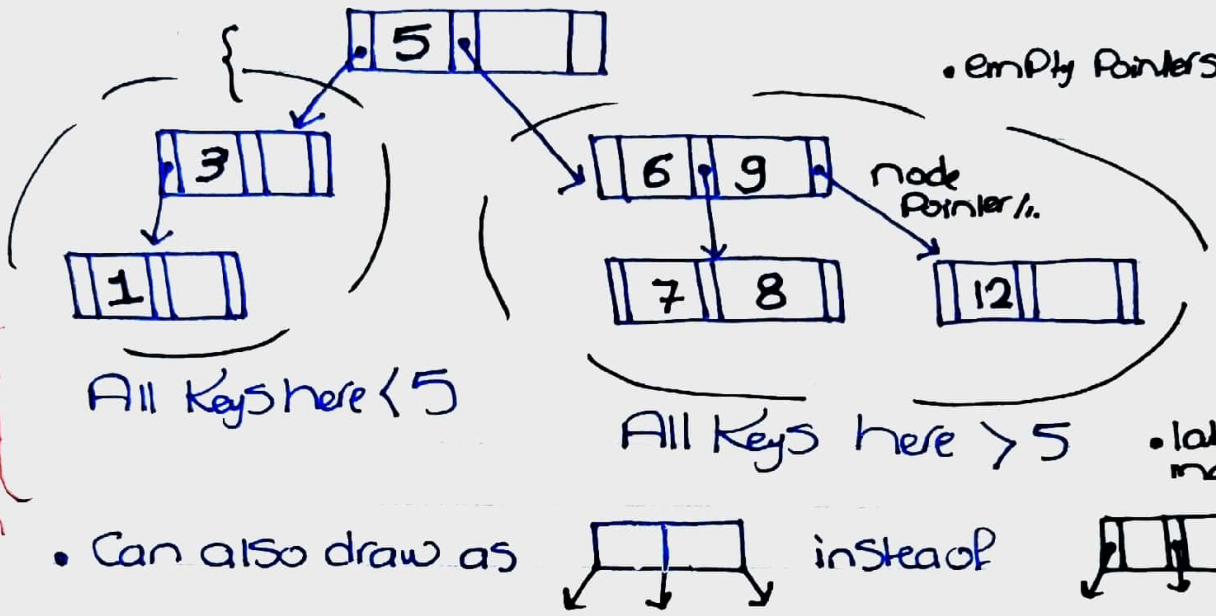
p pointers
p-1 keys.

→ Condition:

For any Key in any node:

- It's greater than any Key in the left Subtree
- It's smaller than any Key in the right Subtree

Example (P = 3) (2 Keys & 3 Ptrs Per □)



• Empty Pointers □ Point to null

Satisfies Search tree condition

• later will make it ≥

• Can also draw as instead of

* B and B+ Trees

→ Variations of Search tree that make it Perfectly balanced (all leaves @ Same level)

• Hence, Insertion and deletion are efficient (and nontrivial) • also ρ

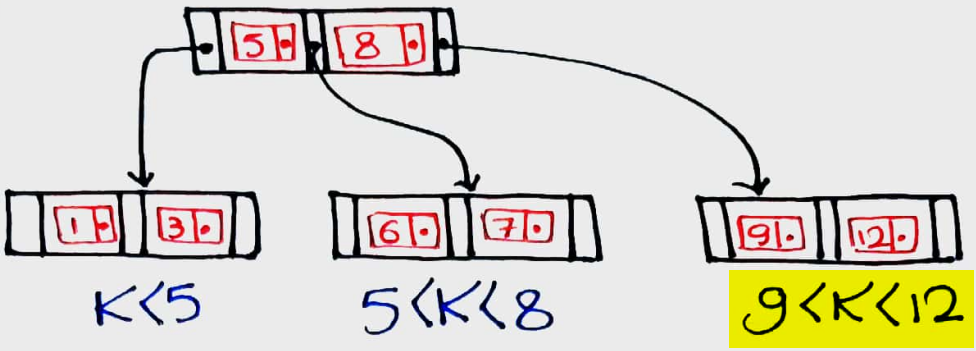
* Each Node Corresponds to a disk block (often has space for new entries)

• Condition: Each node (disk block) must be kept between half-full and completely full

* Most widely used multi-level index.

degen.

- B-Tree

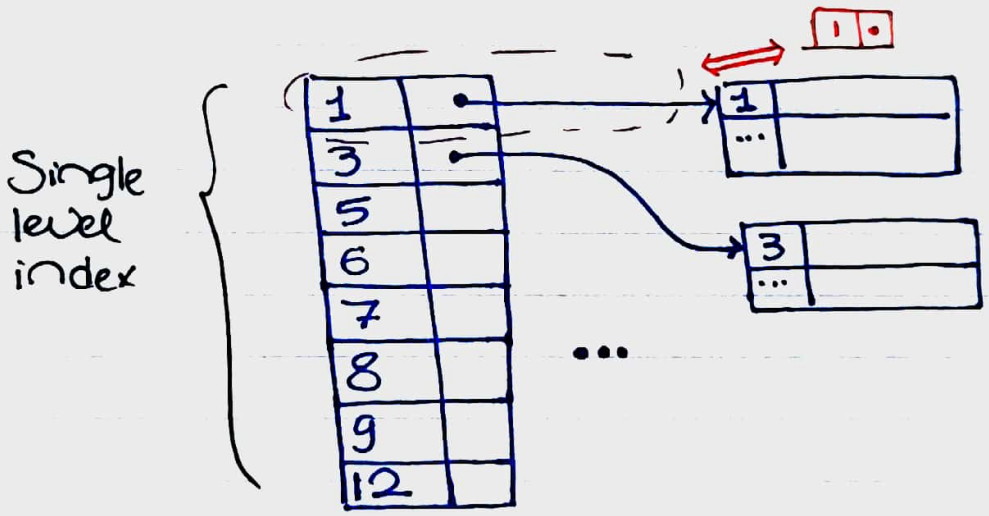


Data Ptr
Node Ptr
Null Ptr

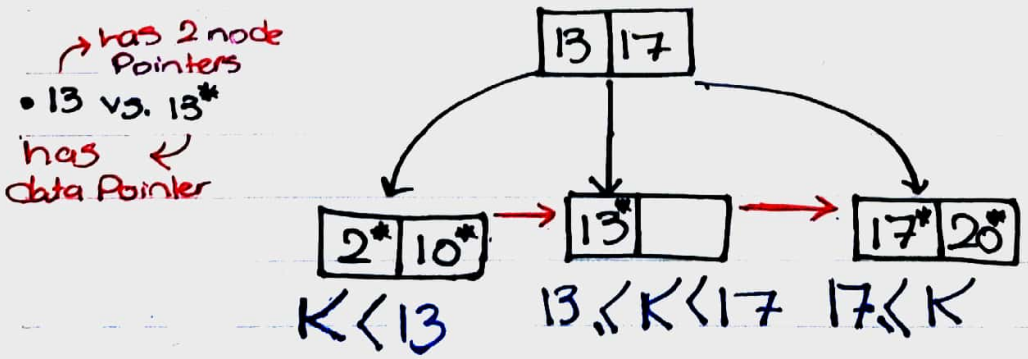
- It's a search tree (we know already)
- Each key is also associated with data Pointer (red dot)

Don't Confuse with node Pointer.

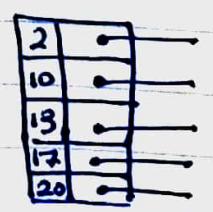
e.g., above tree is an optimization over



- B+ Tree
  - → Restricts that data Pointers must only occur at leaves (only node Pointers about that).
  - Thus, more space to node Pointers; higher branching Factor, higher capacity of search values (i.e., less depth or more search keys than B-trees)



optimization over



- Convention for ≤ might be confused in slides.

- Notice that all leaves are also linked

* In both B and B+ Trees

- Node not full → efficient insertion (just) 
  else may need to split, can affect other levels.

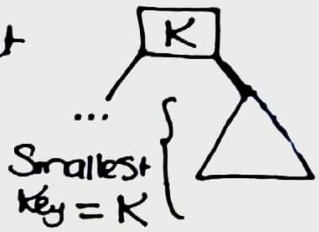
- Node won't become less than half full after deletion → efficient deletion 
  else may need to merge with other nodes.

- We'll focus on B+ Trees

Recap and Notation

- Each leaf node holds up to q (also P_{leaf} or L) entries (at least $\lceil q/2 \rceil$ to satisfy the condition)
- All leaves are at same depth

- Each internal node has P Pointers (& thus P-1 Keys)
  (at least $\lceil P/2 \rceil$ Pointers, $\lceil \frac{P}{2} \rceil - 1$ Keys to satisfy the condition)
- It always holds that



- else, something in the steps is possibly wrong

- Root is initially a leaf node (becomes internal node once it has children; generally)
- Only root can violate the minimum occupancy condition ($\lceil \frac{q}{2} \rceil$ or $\lceil \frac{P}{2} \rceil - 1$) but $P \geq 2$ must hold.

Some Practical tips

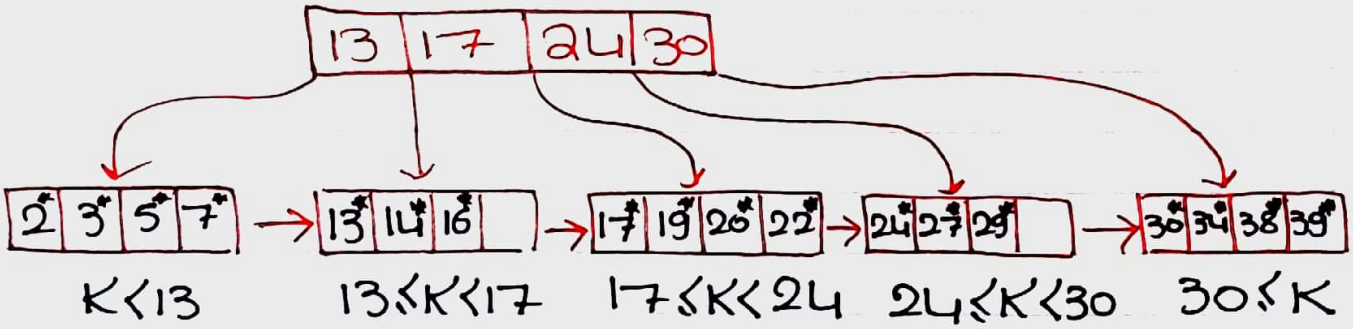
- node = block of data on disk
- To speed up search can buffer 1st one or two levels on main memory

→ most of the time, internal nodes aren't completely full

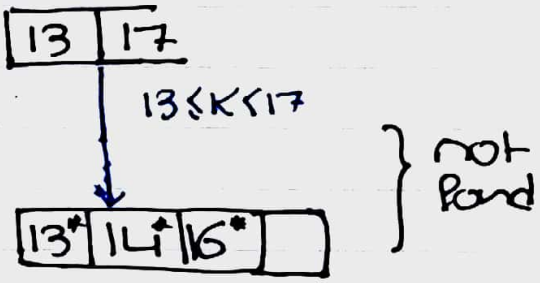
- Big waste for main memory (its small)
- disk is okay with that (big) (efficient in I/del.)

* Will consider searching, insertion & deletion for B+ tree.

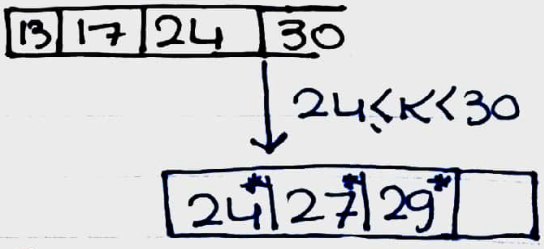
- Range Search & Equality Search

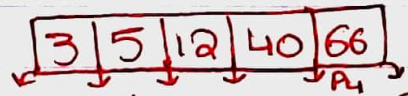


- Equality search for 15



- Equality search for 27



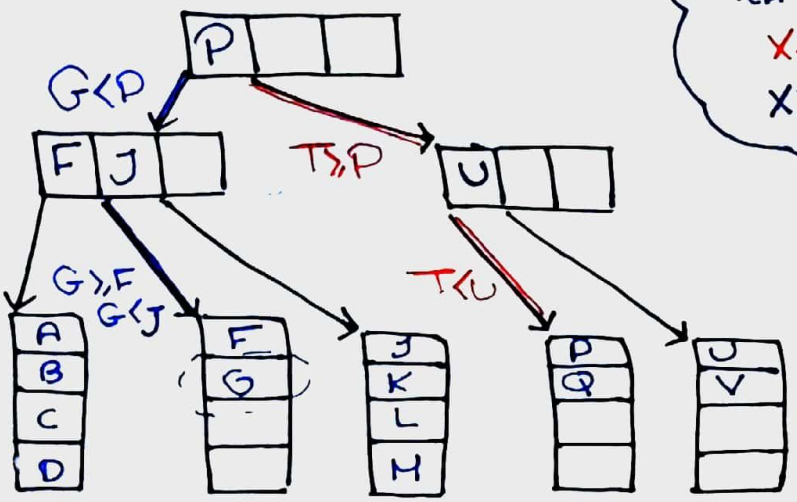
- given 

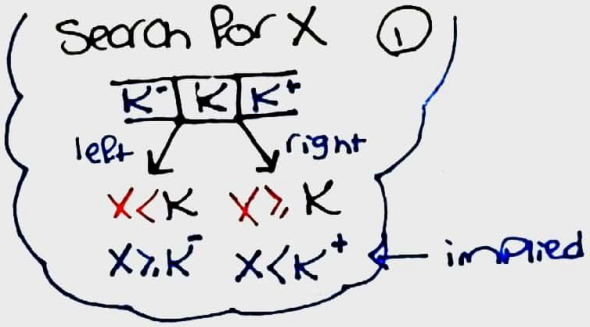
can you from one sight tell which pointer will help us find 55? how about 40?

- Range search for values in [15, 28]
  - → get the leaf block where 15 is expected

  - → loop over keys starting from that block and take all that satisfy condition

Generalizing:





- Search for G
- Search for T
  → not found

- P=9=4

* Can also write an algo. out of this

Start from root & keep taking ptrs until ⇒
- → Check if you should take extreme ptrs of node (X < 1st key or X ≥ last key)
- → use linear or binary search to find when ① is satisfied
- ⇒ Once a leaf is found
  - → use linear or binary search to find the value there (report not found if needed)

* B+ Tree Insertion

- Search for key to find the right block
- If not full, insert it
  e.g., to insert T we have the 4th leaf

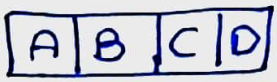
for

to validate:
- Keep in mind leaves are sorted (on both the block and leaves level)

- If full, Create a new Sibling and redistribute then fix Parent Pointers.

e.g., to insert E

→ E<P

→ E<F hence should go in 

• Create a new Sibling (Splitting)

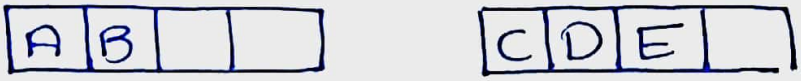
E

• Redistribute such that

q is even { → Middle value is 1st element of new Sibling

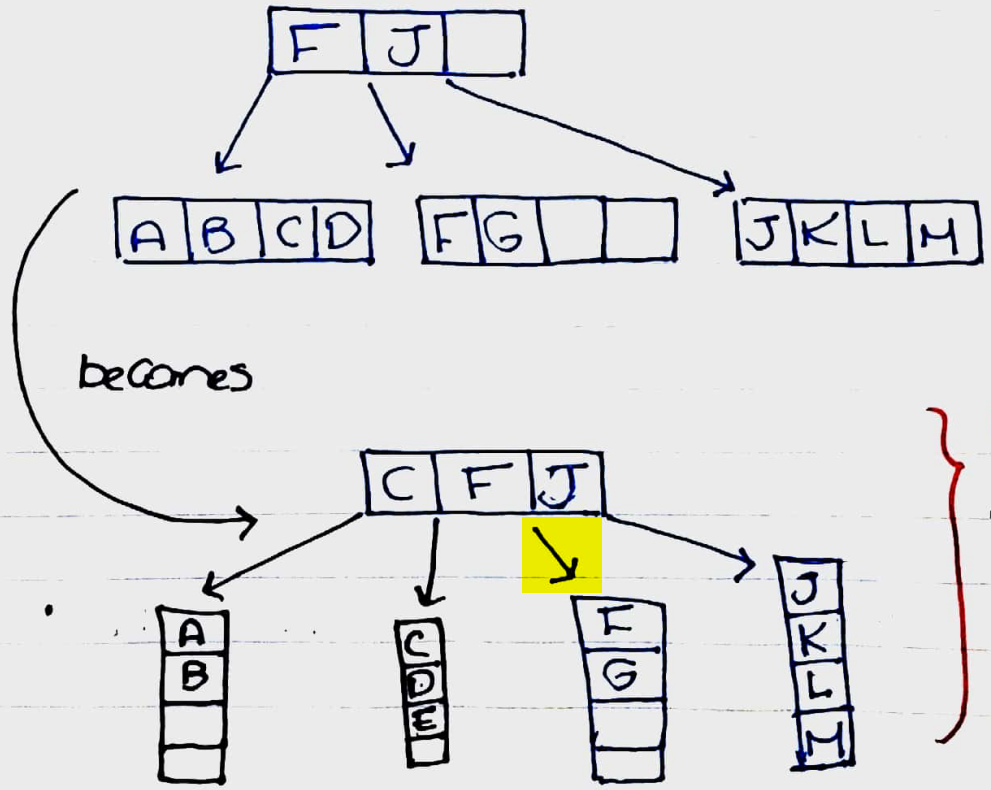
q is odd { → half of the values go to 1st Sibling, 2nd half goes to 2nd Sibling

Thus, in this case



• C is middle value

• Now fix Parent Pointers (adding 1st element of new Sibling 1st)

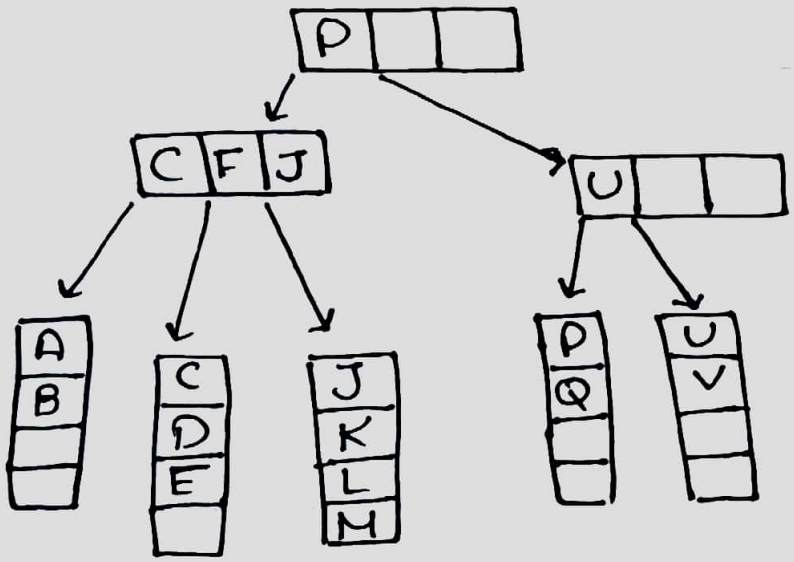


Parent fixed.

• What if the Parent doesn't have space for 1st element of 2nd Sibling?

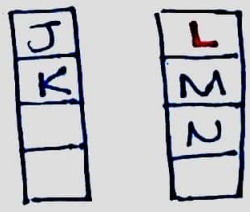
- In that case we'd need to apply a different type of splitting on Parent
(Splitting leaf ≠ Splitting internal node)
  - * what you saw last Page
  - * Happens when leaf has no space
  - * may or may not happen after splitting leaf
    - → In case Parent can't accomodate new Child, will happen.

- To split an internal node
  - → Create a new node (now have two)
  - → move the $\lceil \frac{P}{2} \rceil$th Key to Parent
  - → Redistribute the keys before and after it to 1st and 2nd node respectively



- To insert N
  - → N<P
  - → N>F
    N<J
- goes to 3rd leaf

Parent [C|F|J] has no space for L (CFJL)

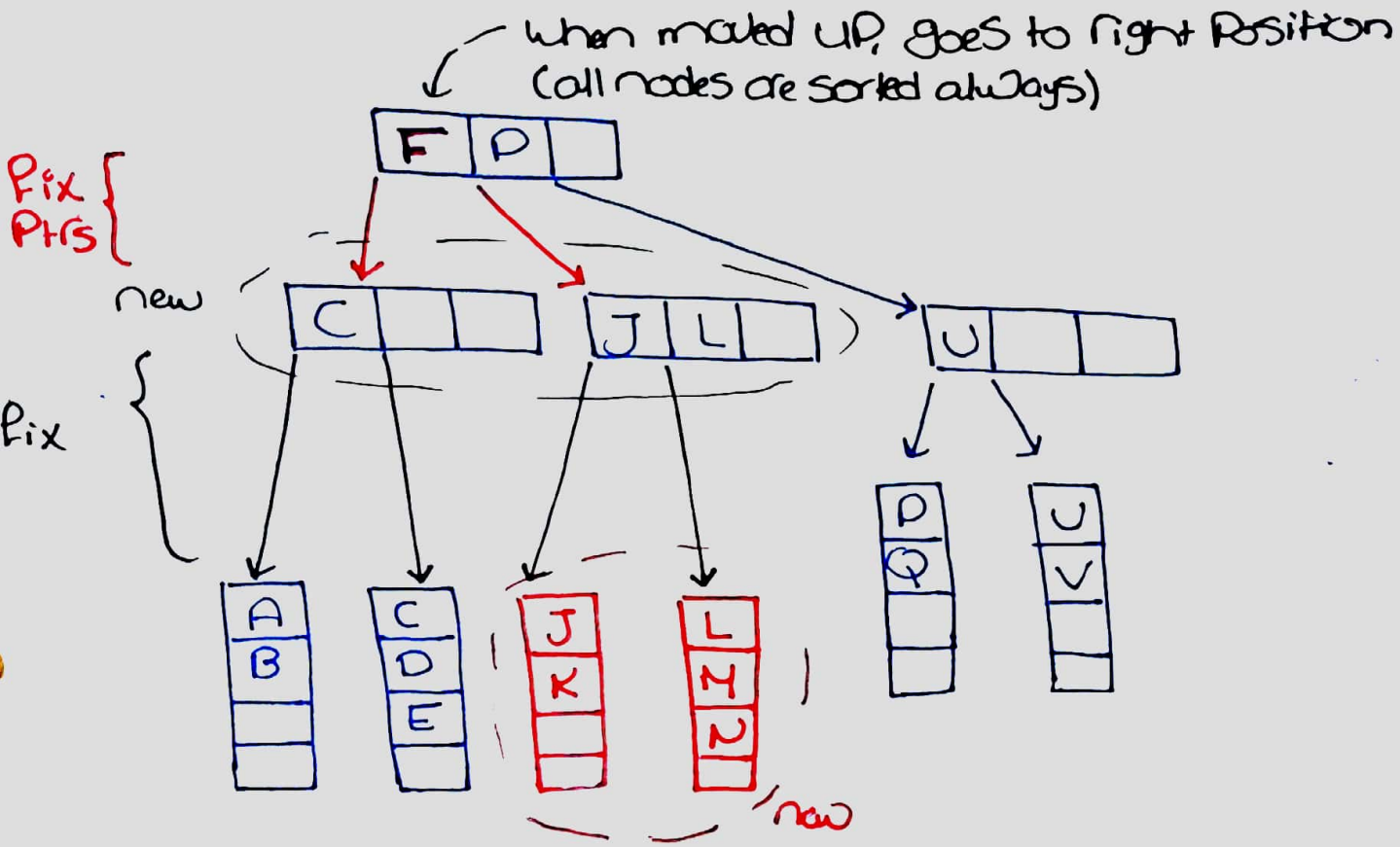


- Split Parent

$P = 4$

$\lceil \frac{P}{2} \rceil_{th} = F$ (goes up)

→ Now fix Parent and children pointers of internal node



→ If the upper level's Parent (was [P |]) can't accomodate the new entry that should move UP, it will be split in the same way. ↻

- Until we reach the root
  - → when the root is split we make the Key that should move UP the only element in the new root.

e.g. [F P S] ← W

→ [P] • new root

FPSW

• $L\frac{P}{2}J_{th} = P$

[F] [S W]

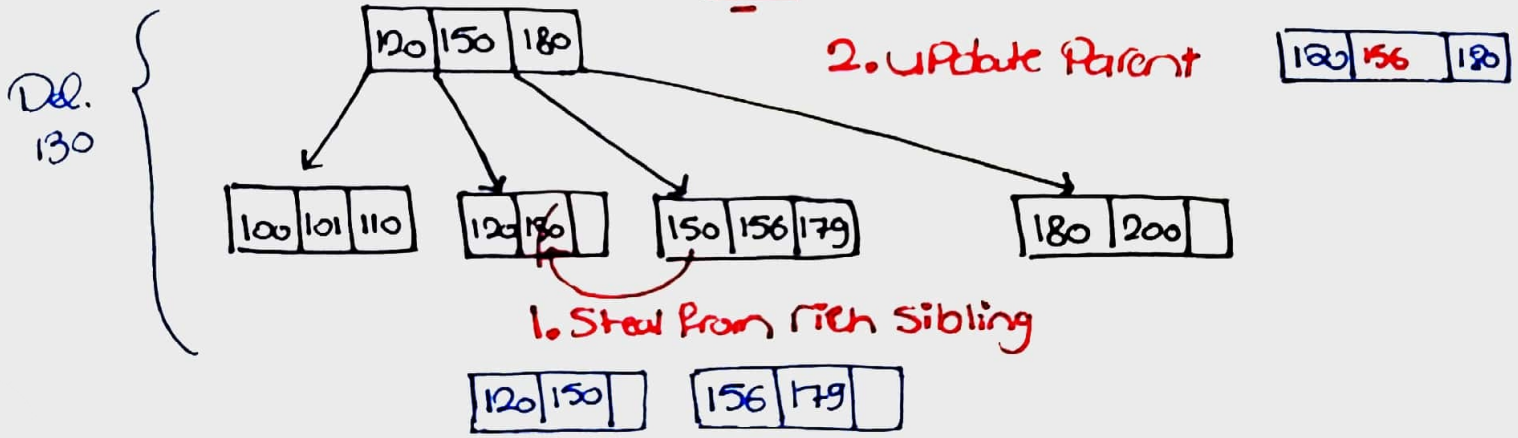
- Notice that depth must increase by 1 whenever root is split

• B+ Tree Deletion

- In B+ Tree
- → Cousins are also Siblings

→ If the leaf node will still have $\lfloor \frac{q}{2} \rfloor$ or more
- Then everything is okay, just delete.

→ Otherwise if after deleting you have less than $\lceil \frac{q}{2} \rceil$ (i.e., $\lceil \frac{q}{2} \rceil - 1$)
- See if Stealing max element from left Sibling or min element from right Sibling doesn't Put them at risk ($\lfloor q/2 \rfloor$).
  → Update Parent Key to match new 1st element



- If both Siblings are Poor (Stealing violates their $\lceil q/2 \rceil$) then merge with one of them
  → guaranteed to fit in a one node
  → Update Parent by removing Key