

CSE 444: Database Internals

Section 3: Indexing

Reminders

- Lab 1 Done!
- Lab 2 released today
 - Will need to run ``git pull upstream lab2`` to get new files
 - Part 1 due April 24th
 - Part 2 due April 29th
- Homework 2 due April 27th
- Today, we will go through indexing examples together

Indexing

- Separate file storing key attribute(s) + pointers (RecordIDs) to records
 - Typically smaller than the data file
- Motivation
 - Fast access to data (less disk I/O)

Motivating Scenario

Consider the following database schema:

Field Name	Data Type	Size on disk
Id (primary key)	Unsigned INT	4 bytes
firstName	Char(50)	50 bytes
lastName	Char(50)	50 bytes
emailAddress	Char(100)	100 bytes

Motivating Scenario

Total records in the database = **5,000,000**

Length of each record = $4+50+50+100 = 204$ bytes

Let the default block size be 1,024 bytes

How many disk blocks are needed to store this data set?

We will have $1024/204 = 5$ records per disk block

No. of blocks needed for the entire table = $5000000/5 = 1,000,000$ blocks

Motivating Scenario

Suppose you want to find the person with a particular **id** (say 5000)

Assume data file sorted on primary key

What is the best way to do so?

Motivating Scenario

Linear Search

No. of block accesses = $1000000/2$
= **500,000 on avg**

Binary Search

No. of block accesses = $\log_2 1000000 = 19.93 = \mathbf{20}$

Motivating Scenario

Now, suppose you want to find the person having **firstName = 'John'**

Here, the column isn't sorted and does not hold an unique value.

What is the best way to do search for the records?

Motivating Scenario

Solution: Create an index on the **firstName** column

The schema for an index on **firstName** is:

Field Name	Data Type	Size on disk
firstName	Char(50)	50 bytes
(record pointer)	Special	4 bytes

Motivating Scenario

Total records in the database = **5,000,000**

Length of each index record = $4+50 = 54$ bytes

Let the default block size be **1,024 bytes**

Therefore,

We will have $1024/54 = 18$ records per disk block

Also, No. of blocks needed for the entire table =
 $5000000/18 = 277,778$ blocks

Motivating Scenario

Now, a binary search on the index will result in $\log_2 277778 = 18.08 =$ **19 block accesses**.

Also, to find the address of the actual record, which requires a further block access to read, bringing the total to $19 + 1 =$ **20 block accesses**.

Thus, indexing results in a much better performance as compared to searching the entire database.

Indexes

Useful for search query / range query / joins

Tweet Example:

Tweets(tid, user, time, content)

Tweet Relation in a Sequential File

tid	user	time	content
10	1	05:03:00	"....."
20	2	12:05:07	"....."
30	2	18:12:00	"....."
40	3	00:16:13	"....."
50	4	10:10:13	"....."
60	1	04:09:07	"....."
70	2	12:08:34	"....."
80	4	11:08:09	"....."

1 record

1 page

- File is sorted on "tid"

Index Classification

- **Primary/secondary**
 - Primary = determines the location of indexed records on disk
 - Secondary = cannot reorder data, does not determine data location
- **Dense/sparse**
 - Dense = every key in the data appears in the index
 - Sparse = the index contains only some keys
- **Clustered/unclustered**
 - Clustered = records close in index are close in data
 - Unclustered = records close in index may be far in data

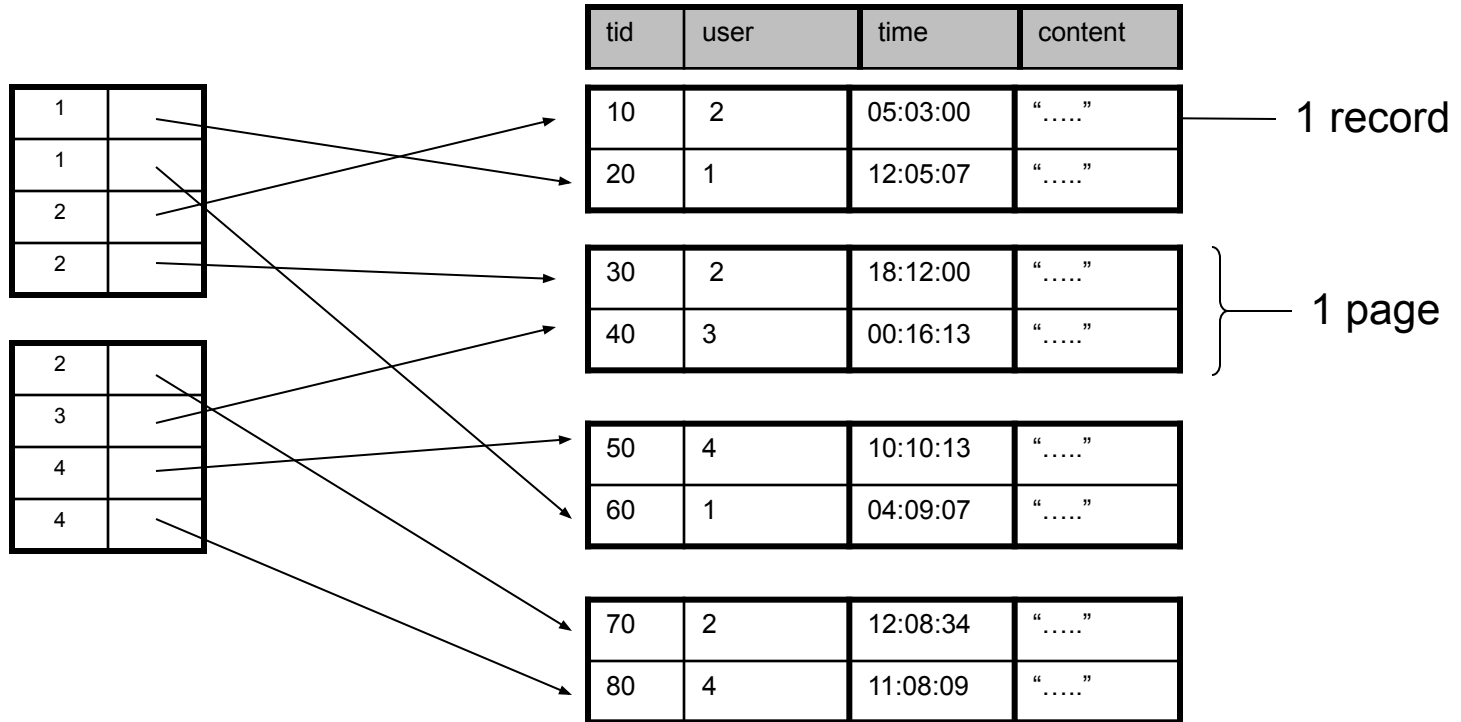
Ex1. Draw a secondary dense index on “user”

tid	user	time	content
10	2	05:03:00	“.....”
20	1	12:05:07	“.....”
30	2	18:12:00	“.....”
40	3	00:16:13	“.....”
50	4	10:10:13	“.....”
60	1	04:09:07	“.....”
70	2	12:08:34	“.....”
80	4	11:08:09	“.....”

1 record

1 page

Ex1. Secondary Dense Index (user)



- **Dense:** an "index key" (not database key) for every database record
- **Secondary:** cannot reorder data, does not determine data location
- Also, **Unclustered:** records close in index may be far in data

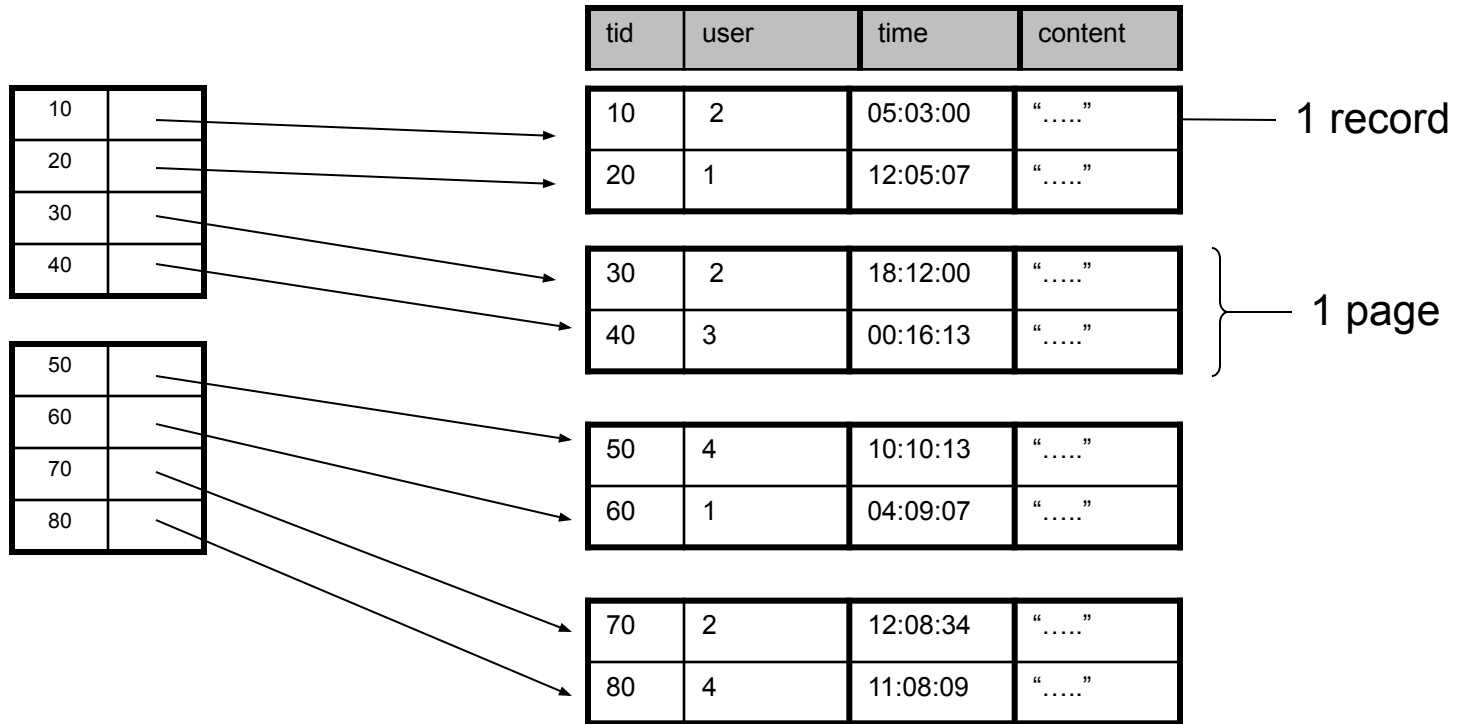
Ex2. Draw a primary dense index on “tid”

tid	user	time	content
10	1	05:03:00	“.....”
20	2	12:05:07	“.....”
30	2	18:12:00	“.....”
40	3	00:16:13	“.....”
50	4	10:10:13	“.....”
60	1	04:09:07	“.....”
70	2	12:08:34	“.....”
80	4	11:08:09	“.....”

1 record

1 page

Ex2. Primary Dense Index (tid)



- **Dense:** an “index key” for every database record
 - (In this case) every “database key” appears as an “index key”
- **Primary:** determines the location of indexed records
- Also, **Clustered:** records close in index are close in data

Improve from Primary Clustered Index?

Clustered Index can be made Sparse
(normally one page per key)

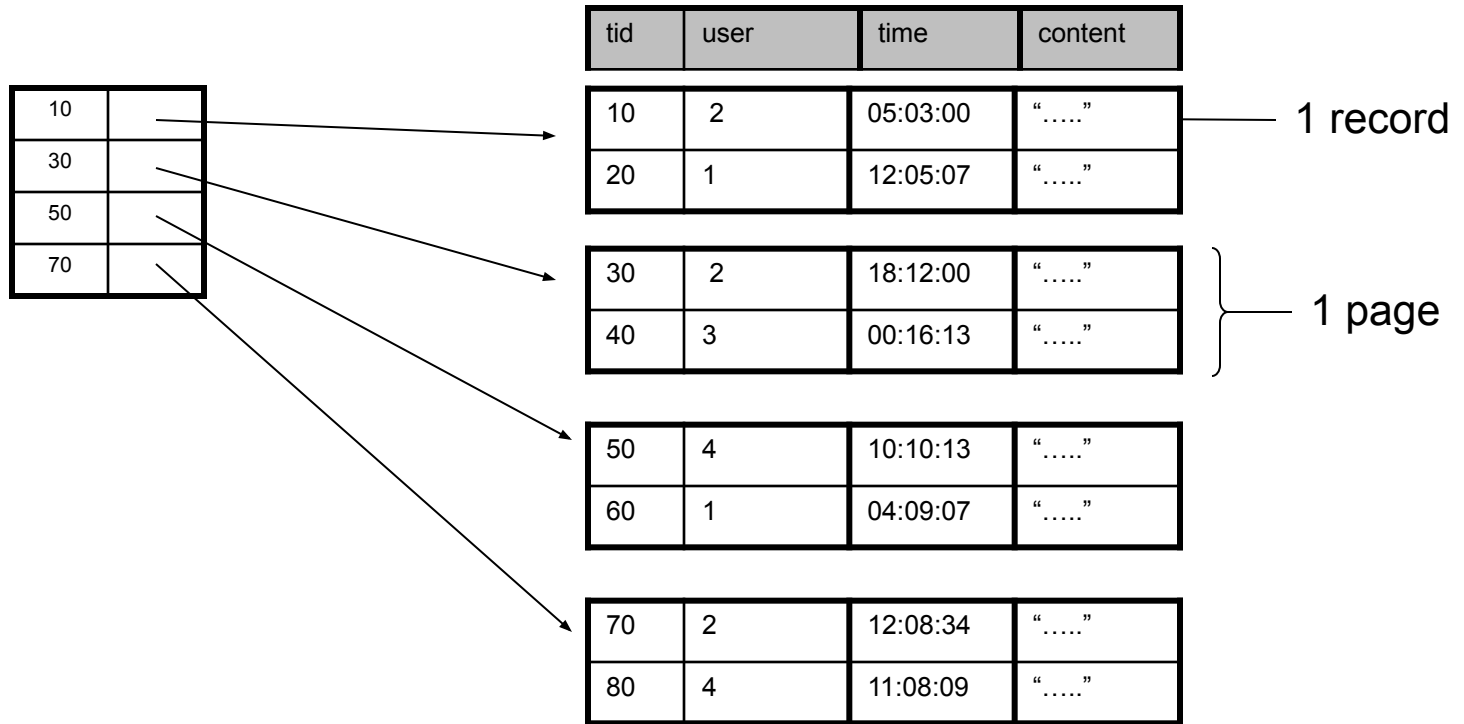
Ex3. Draw a primary sparse index on “tid”

tid	user	time	content
10	2	05:03:00	“.....”
20	1	12:05:07	“.....”
30	2	18:12:00	“.....”
40	3	00:16:13	“.....”
50	4	10:10:13	“.....”
60	1	04:09:07	“.....”
70	2	12:08:34	“.....”
80	4	11:08:09	“.....”

1 record

1 page

Ex3. Primary Sparse Index (tid)

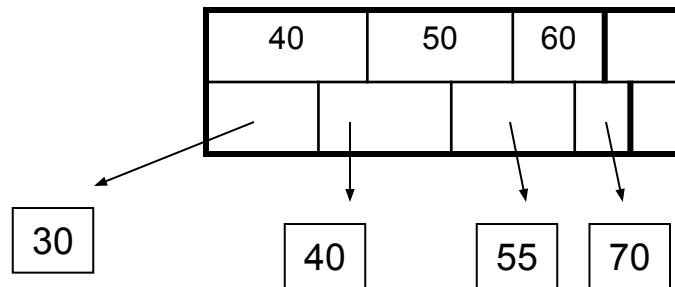


- Only one index file page instead of two

B+ trees

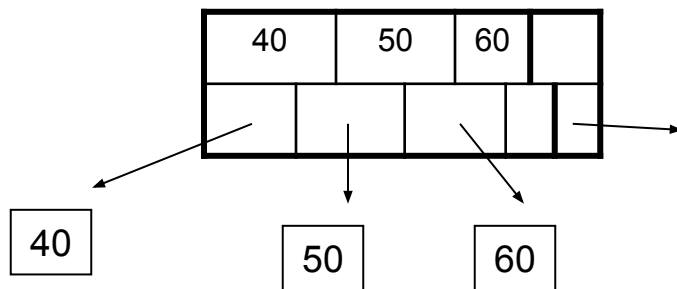
Insertions and Deletion in a B+ tree

- Note: the $<$, \geq assumptions in this class:



Internal node:

- Left pointer from key = k: to keys $< k$
- Right pointer: to keys $\geq k$



Leaf node:

- Left pointer from key = k: to the block containing data with value k in that attribute
- Last remaining pointer on right: To the next leaf on right

Insertions and Deletion in a B+ tree

- Note: when a leaf is split, the middle key is copied to the new leaf on **right** (and also inserted in parent)
 - Since we assumed the **right pointer** from $\text{key} = k$ points to keys $\geq k$
- Note: when an internal node is split, we do not need to copy the middle key to the right, only insert it in parent
 - Use the left pointer of the new right internal node
- Some examples....

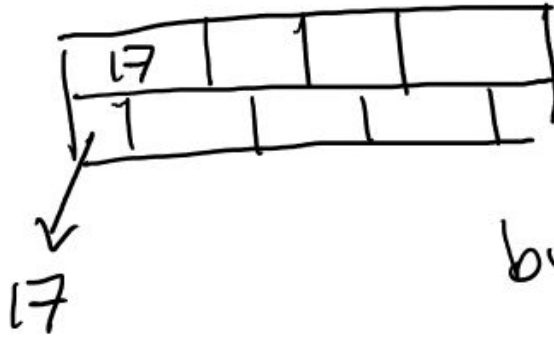
Problem 1:

B+ tree insertion and deletion

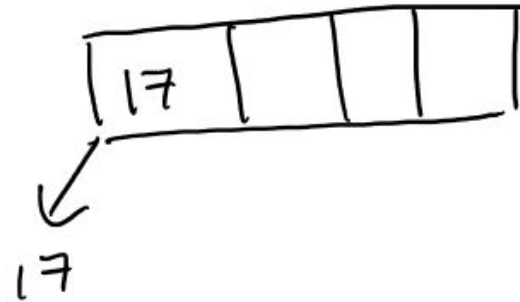
- Start with an empty B+ tree, $d=2$
- Insert 17, 3, 25, 95, 8, 57, 69
- Then insert 29, 91, 78, 80, 92, 99, 97

Insertions

17

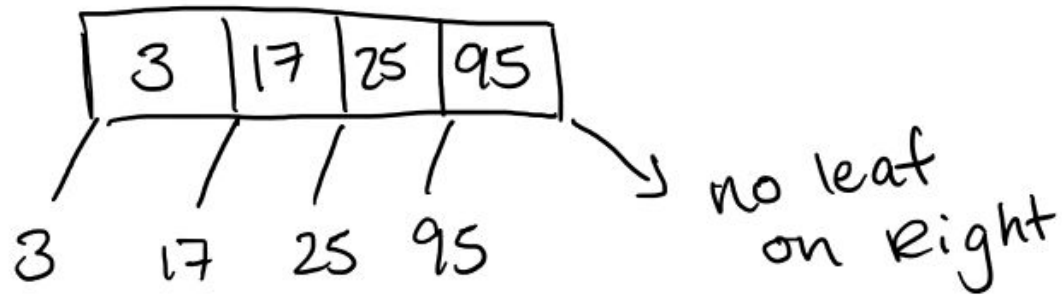


but we will use:

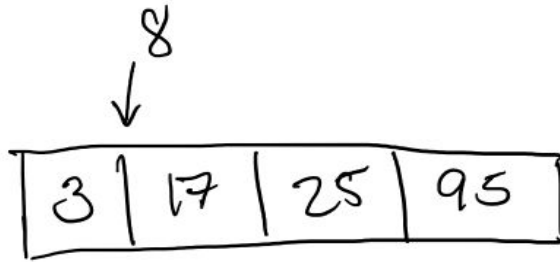


Insertions

3, 25, 95

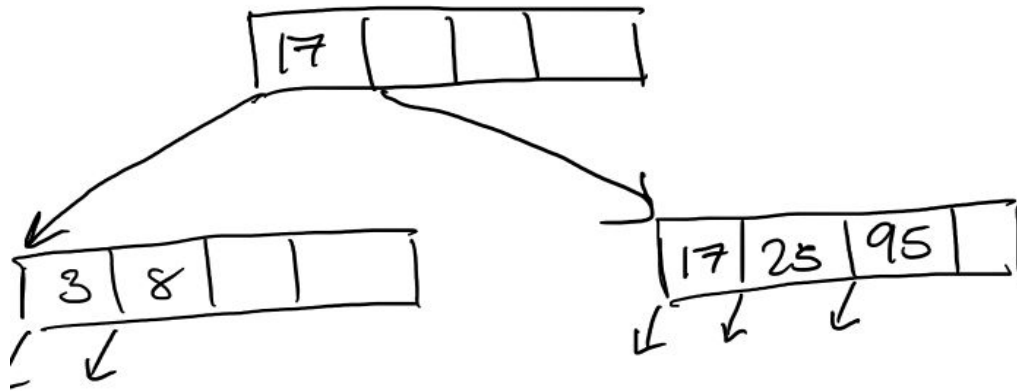


Insertions



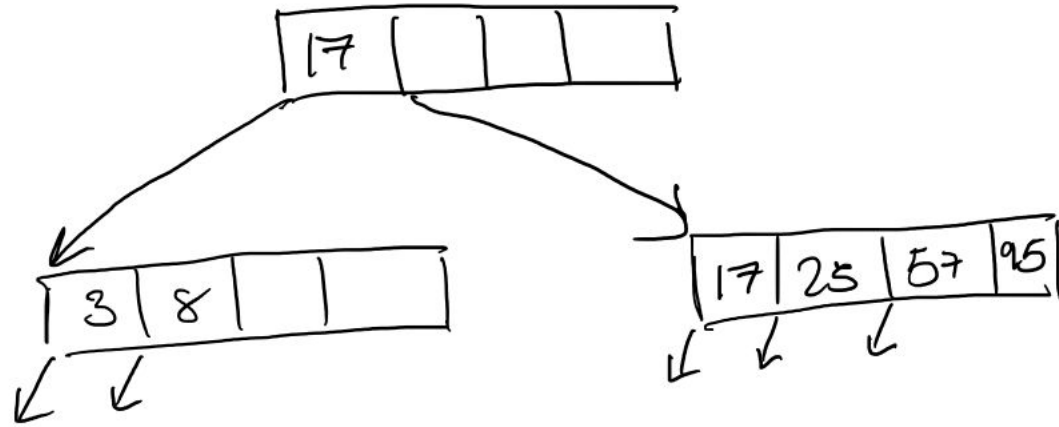
⋮

Split based on middle (17)



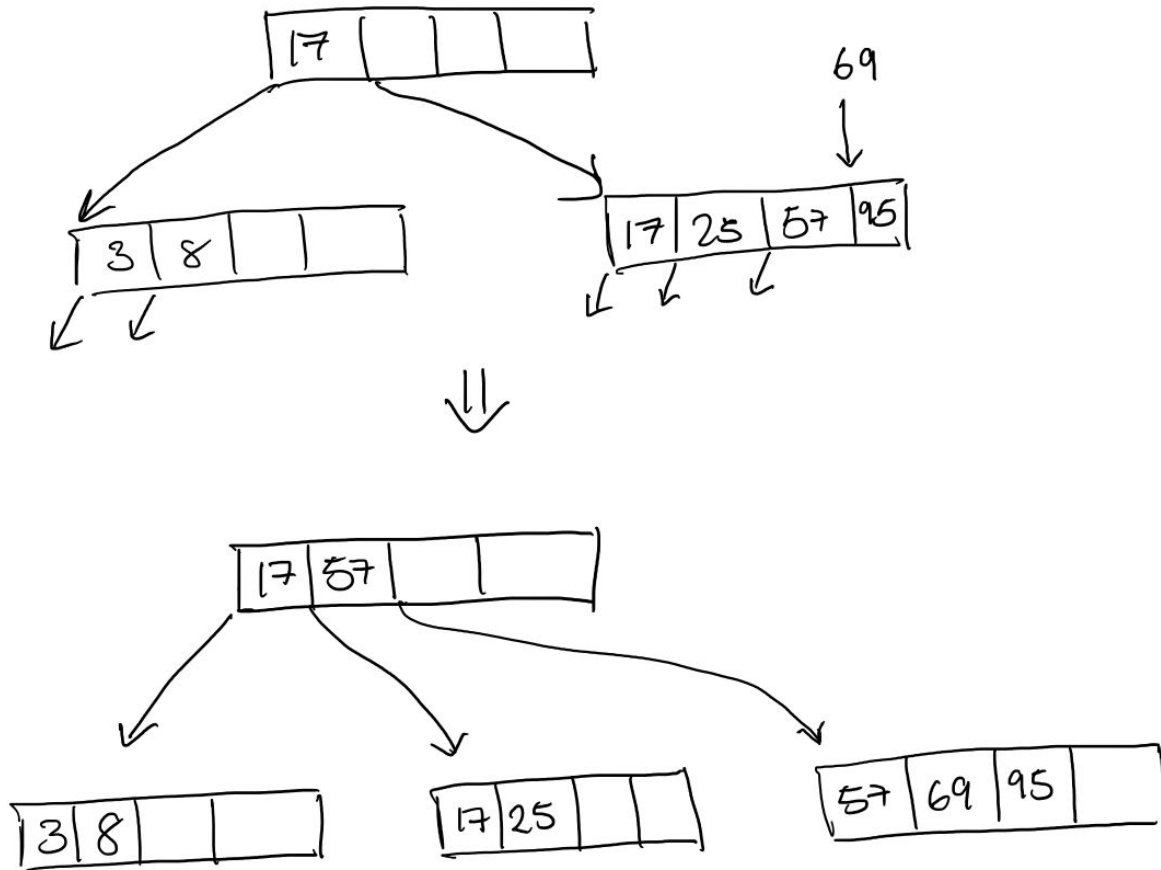
Insertions

57



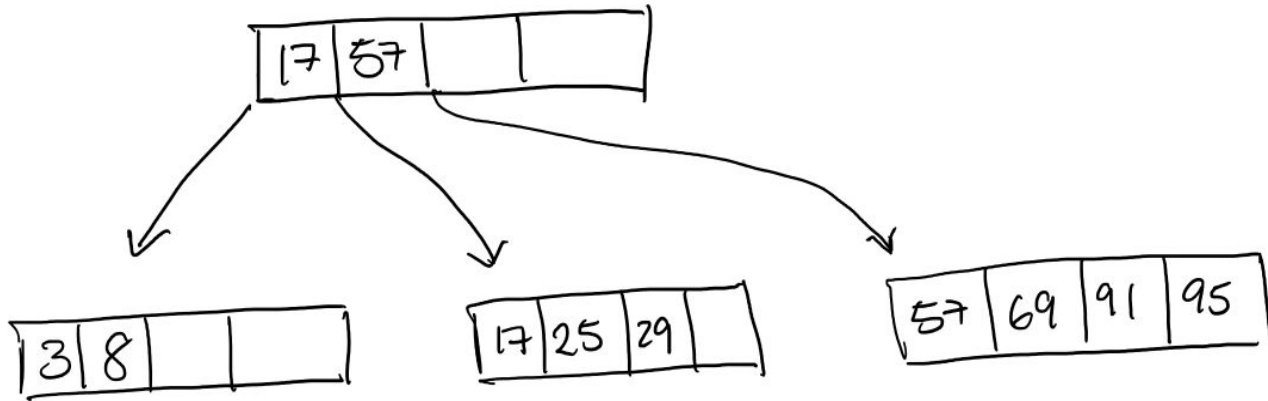
Insertions

69



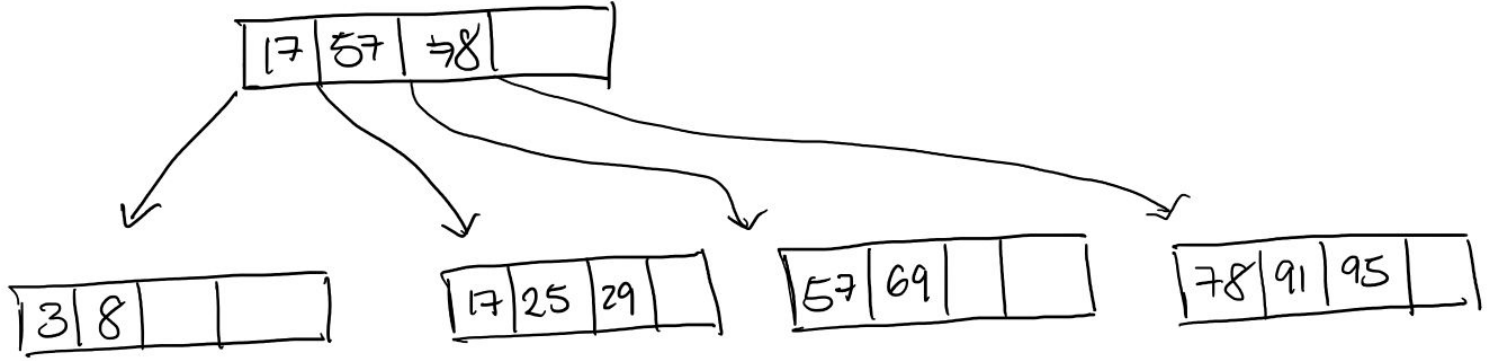
Insertions

29, 91



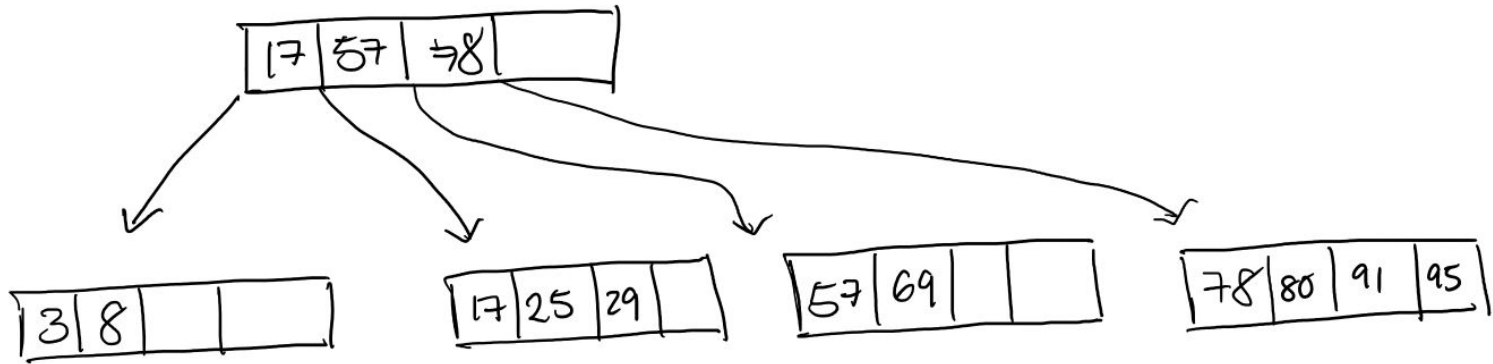
Insertions

78



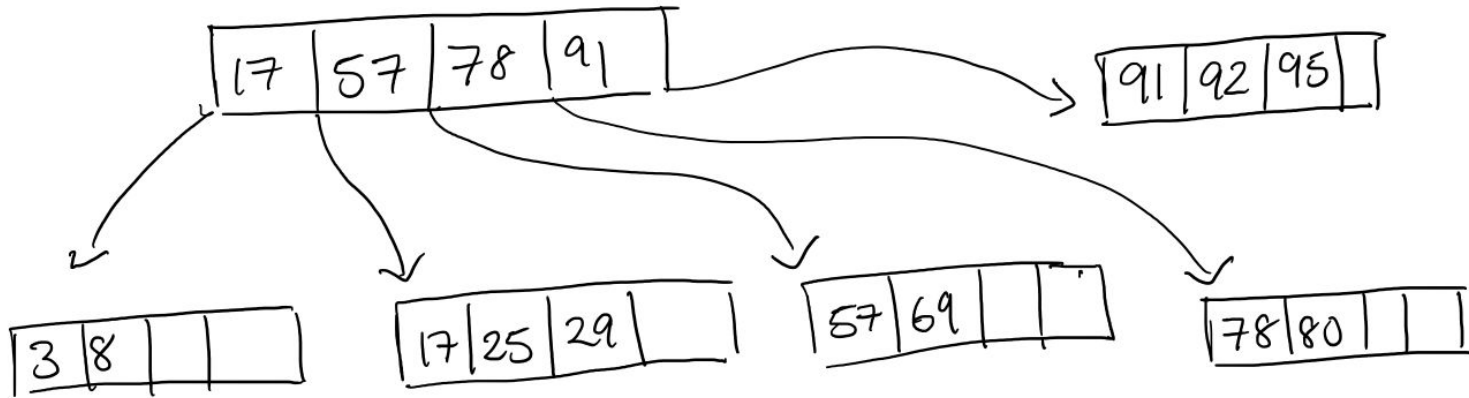
Insertions

80



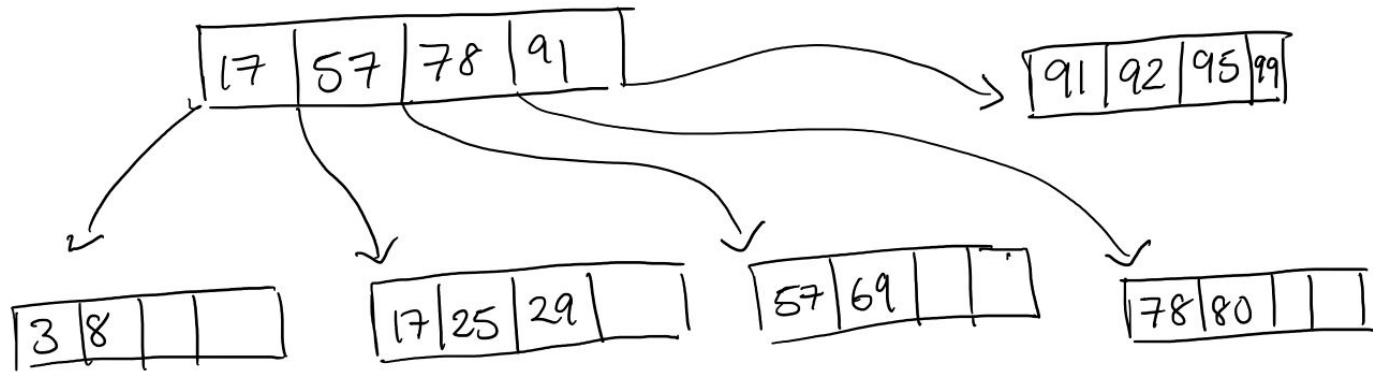
Insertions

92



Insertions

99

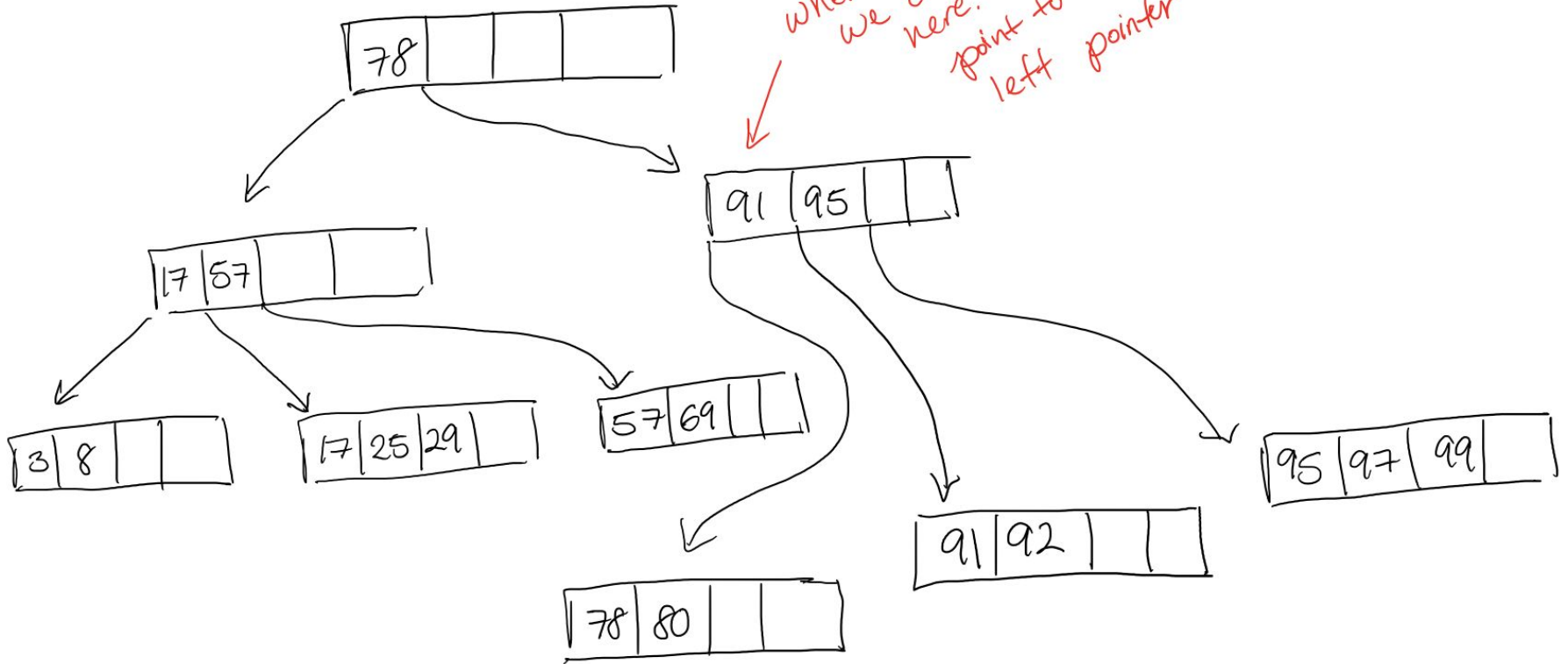


Insertions

97

need to split Root!

when splitting Root Node
we don't move 78
here. we can simply
point to it using
left pointer



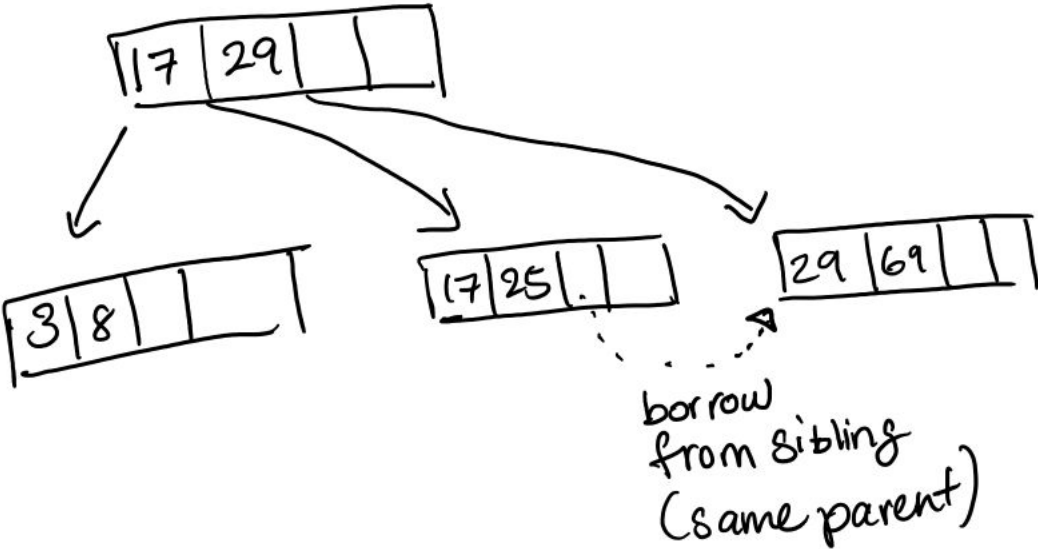
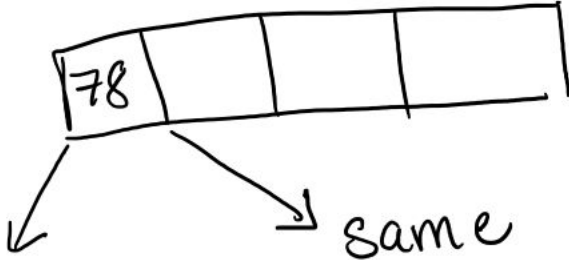
Problem 1:

B+ tree insertion and deletion

- Now delete all nodes in the following order:
57, 3, 99, 29, 17, 25, 95, 8, 78, 92, 69, 97, 91

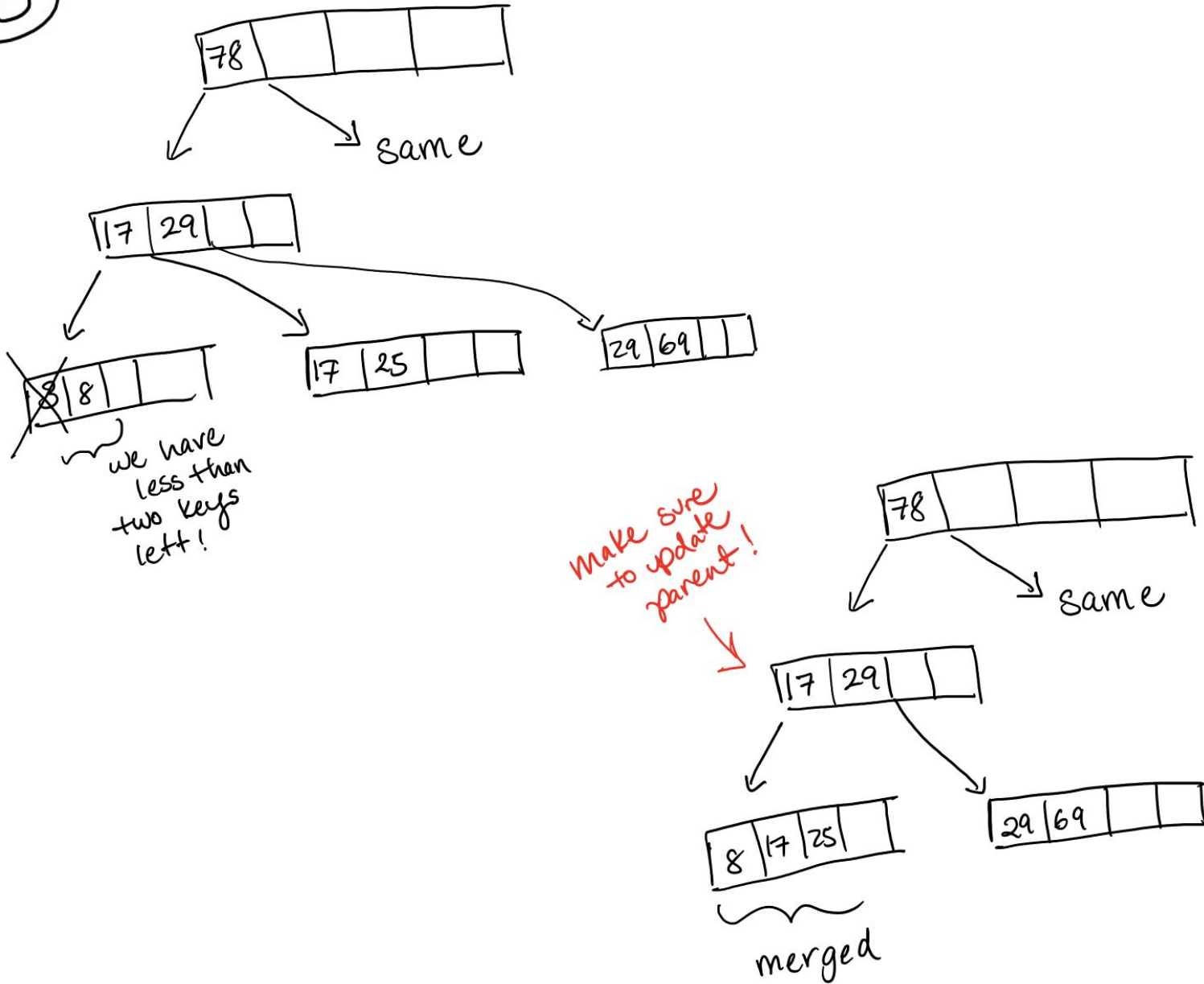
Deletions

57



3

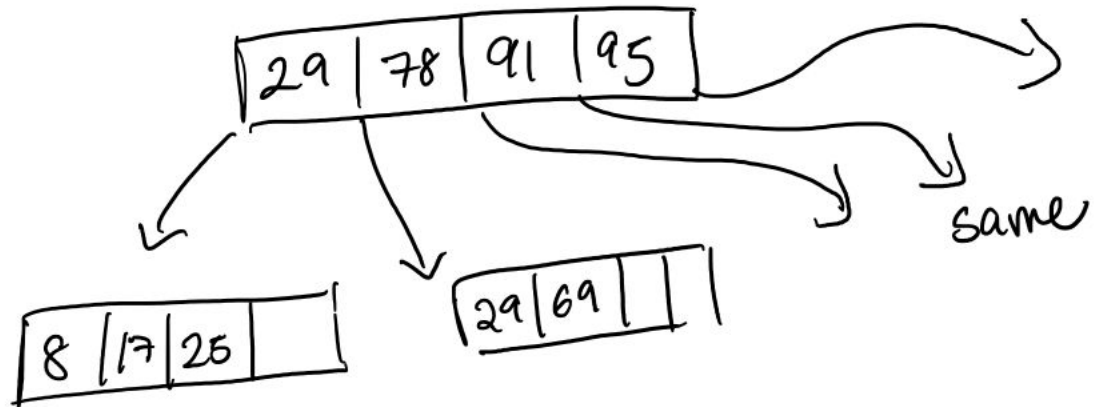
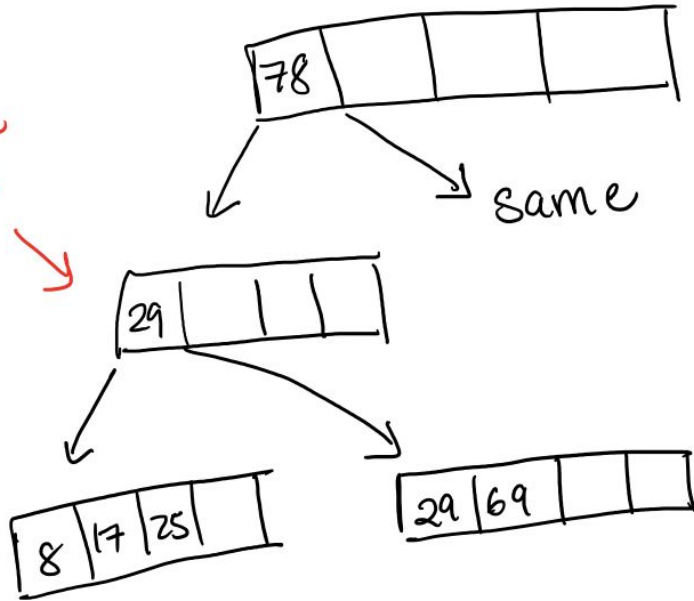
Deletions



3

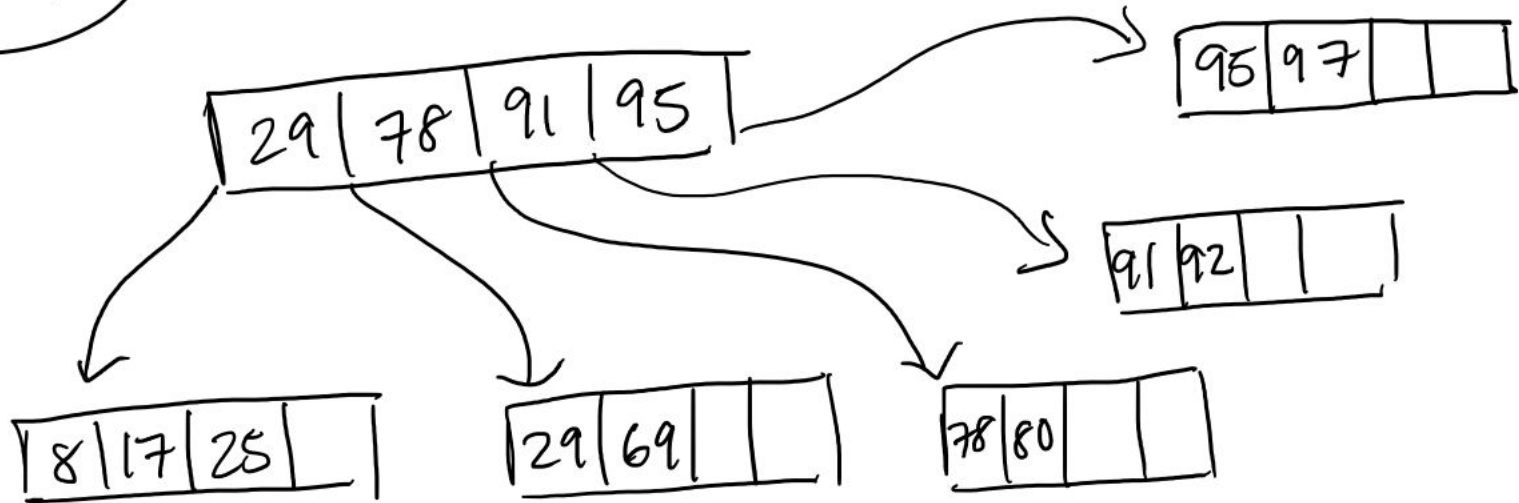
Deletions (continued for 3)

now <2
keys



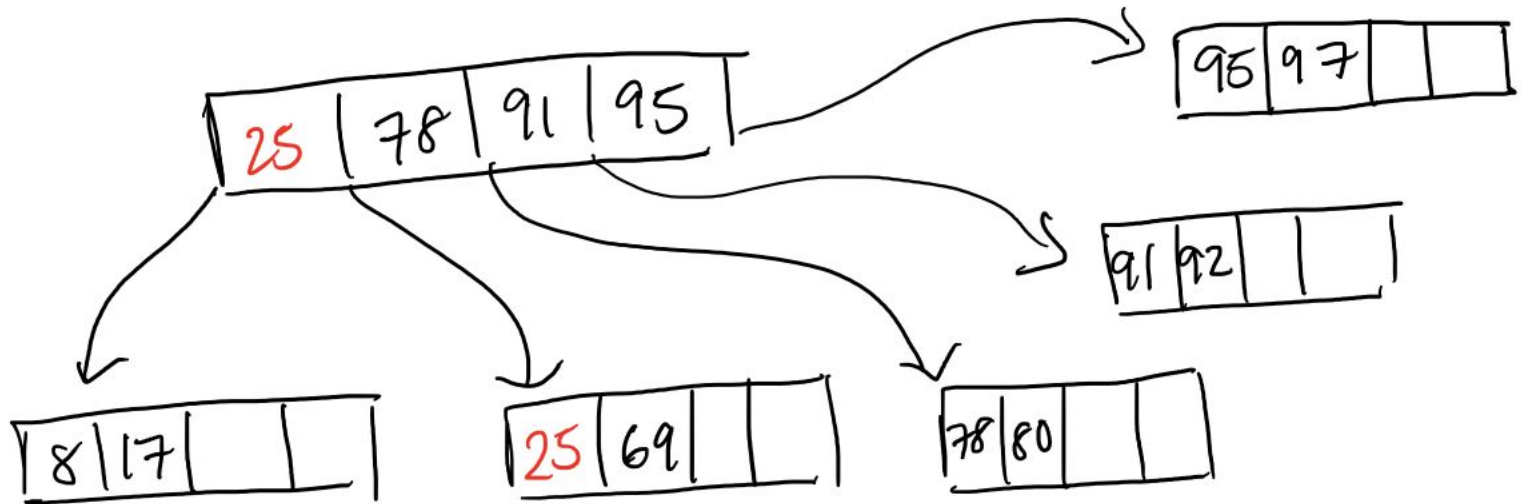
Deletions

99



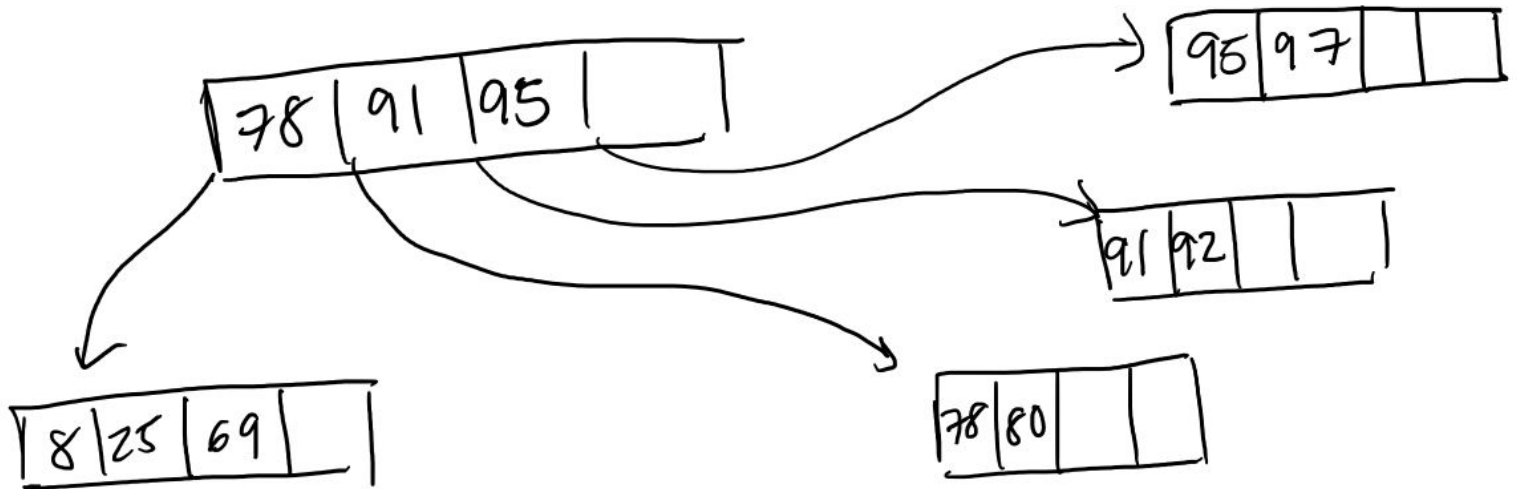
Deletions

29



Deletions

(17)



when merging,
delete separating key
in parent!

- Note: next few slides are older versions of the previous slides

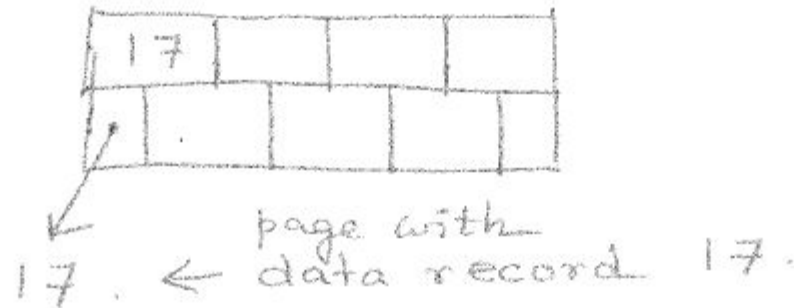
Problem 1:

B+ tree insertion and deletion

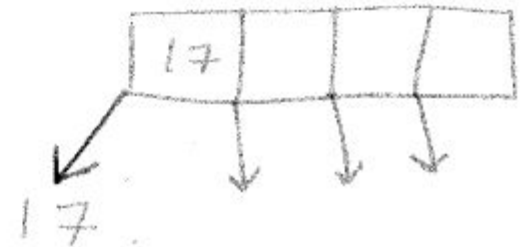
- Start with an empty B+ tree, $d=2$
- Insert 17, 3, 25, 95, 8, 57, 69
- Then insert 29, 91, 78, 80, 92, 99, 97

Problem 1: B+ tree insertion and deletion

Insert 17

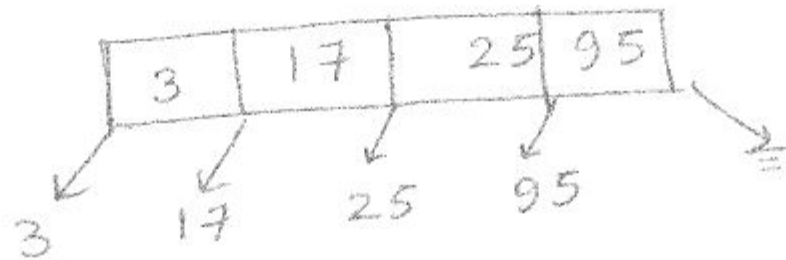


We are going to write it as



Problem 1: B+ tree insertion and deletion

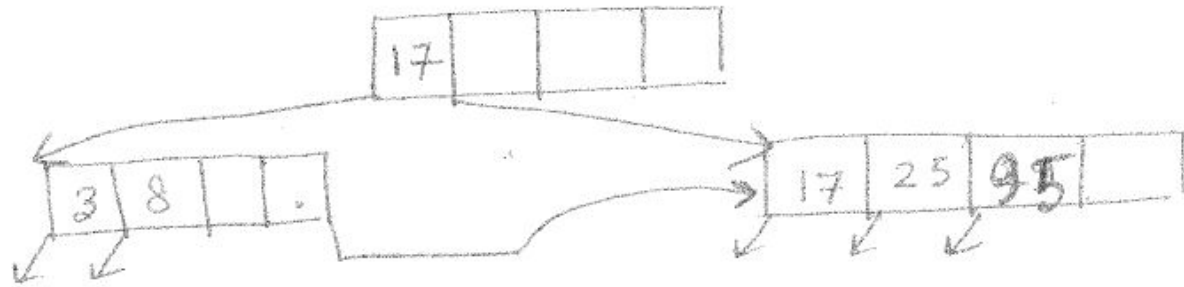
Insert 3, 25, 95



no
next leaf
on right.

Problem 1: B+ tree insertion and deletion

Insert 8

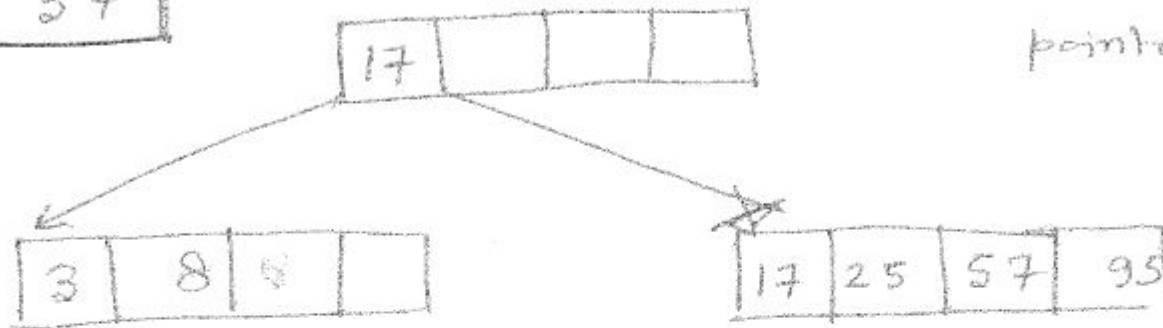


Note: A Leaf is being split
Copy middle key to right leaf

(Later :- copying the key at the middle
is not needed when an internal
node is split)

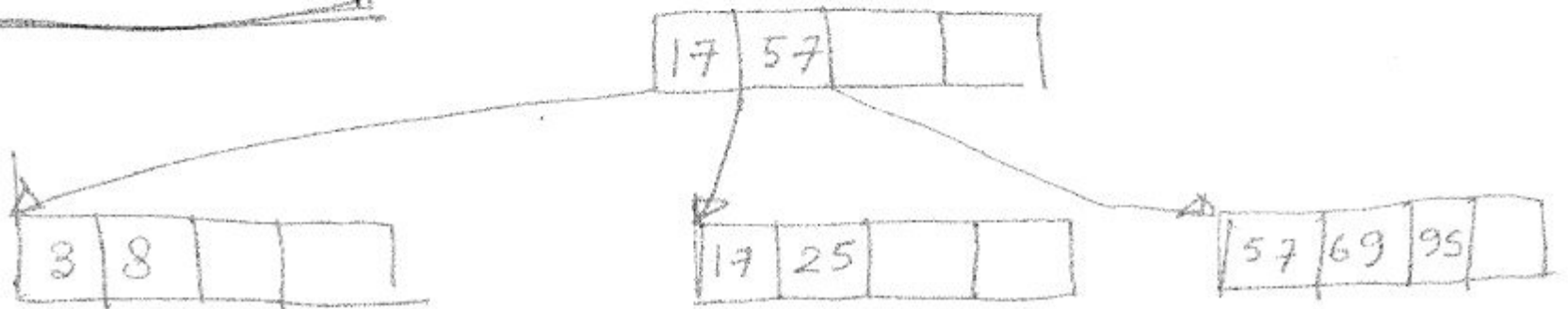
Problem 1: B+ tree insertion and deletion

Insert 57



[omitting pointers
to data &
pointer to right
next leaf]

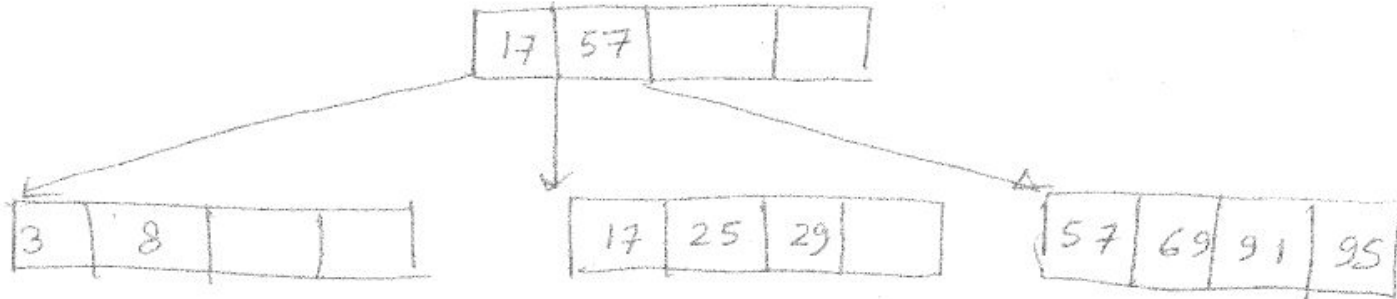
Insert 69



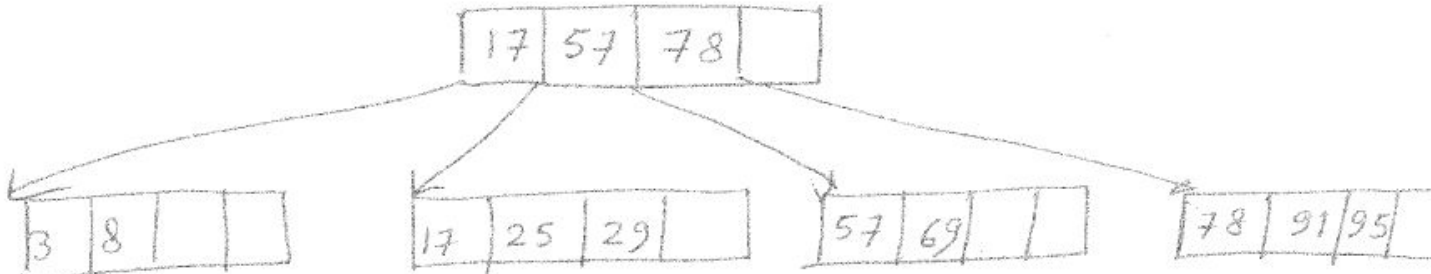
Problem 1:

B+ tree insertion and deletion

Insert 29, 91

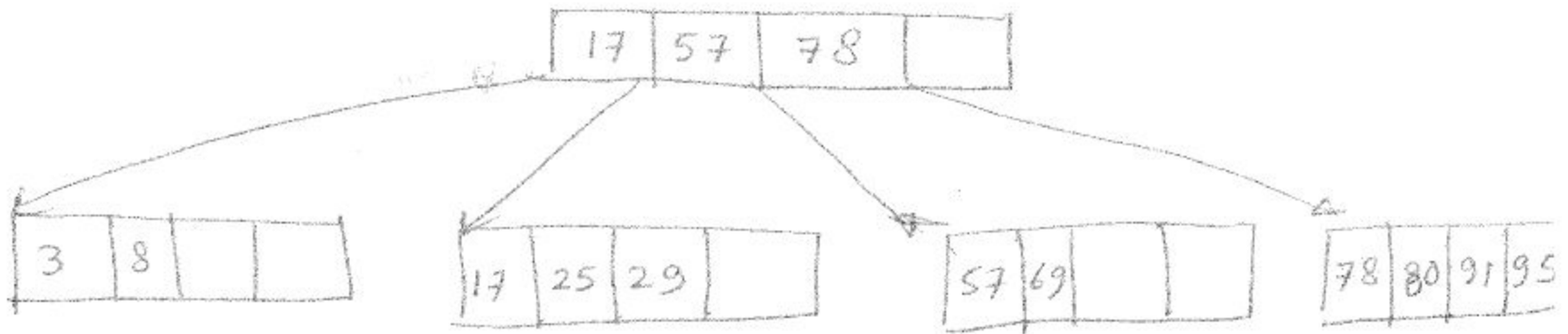


Insert 78

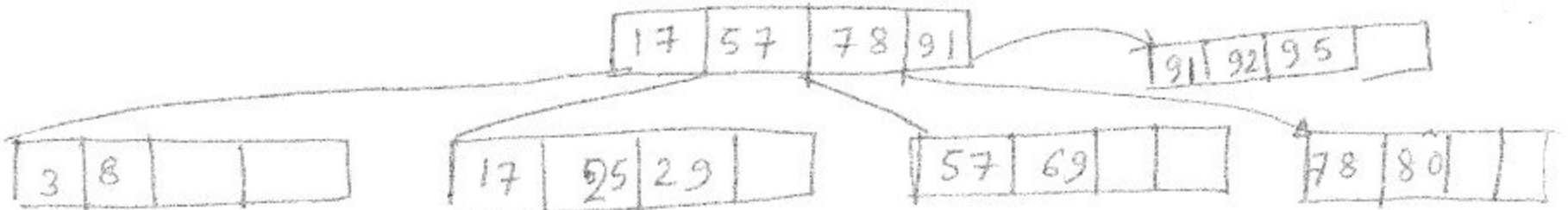


Problem 1: B+ tree insertion and deletion

Insert 80



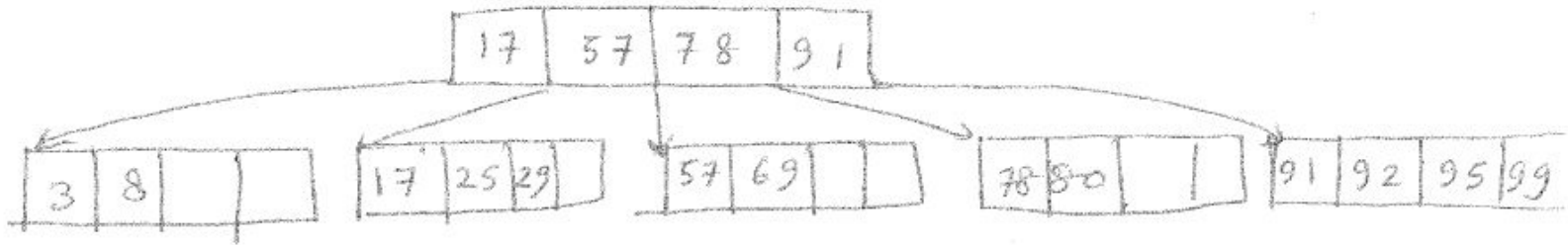
Insert 92



Problem 1:

B+ tree insertion and deletion

Insert 99

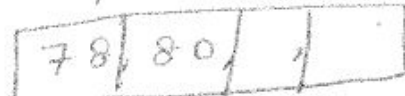


Insert 97

Need to split root, height increases!

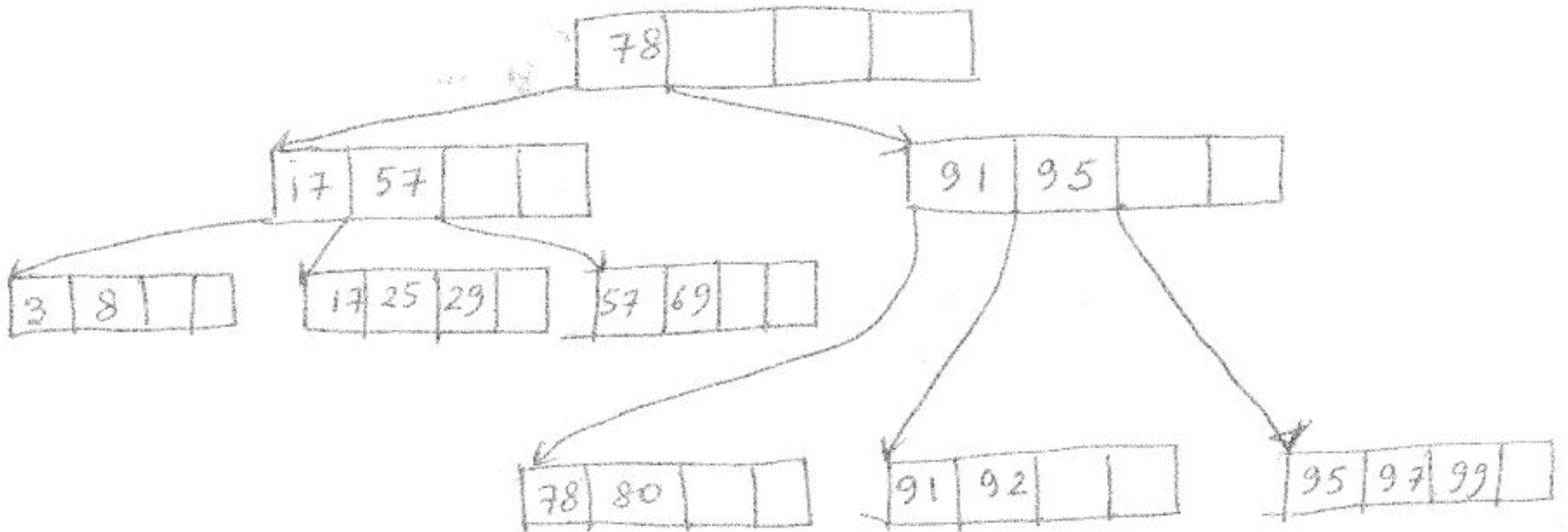
Note :- 78 is not being copied to right new node, unlike leaves.

why? we have left pointer from 91 which can point to



Problem 1:

B+ tree insertion and deletion



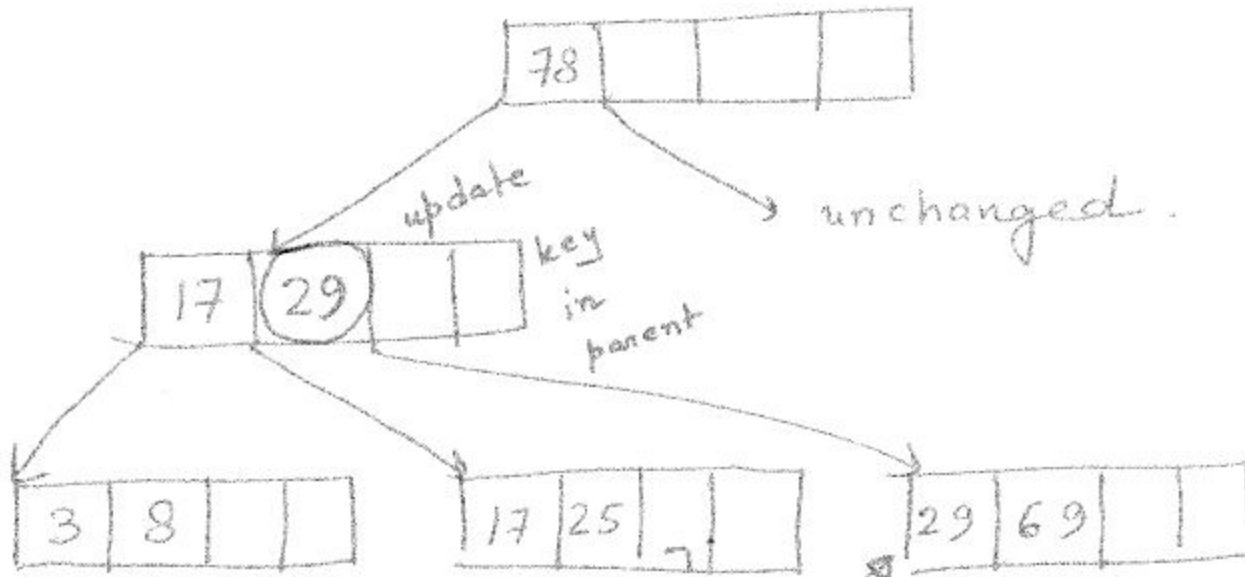
Problem 1:

B+ tree insertion and deletion

- Now delete all nodes in the following order:
57, 3, 99, 29, 17, 25, 95, 8, 78, 92, 69, 97, 91

Problem 1: B+ tree insertion and deletion

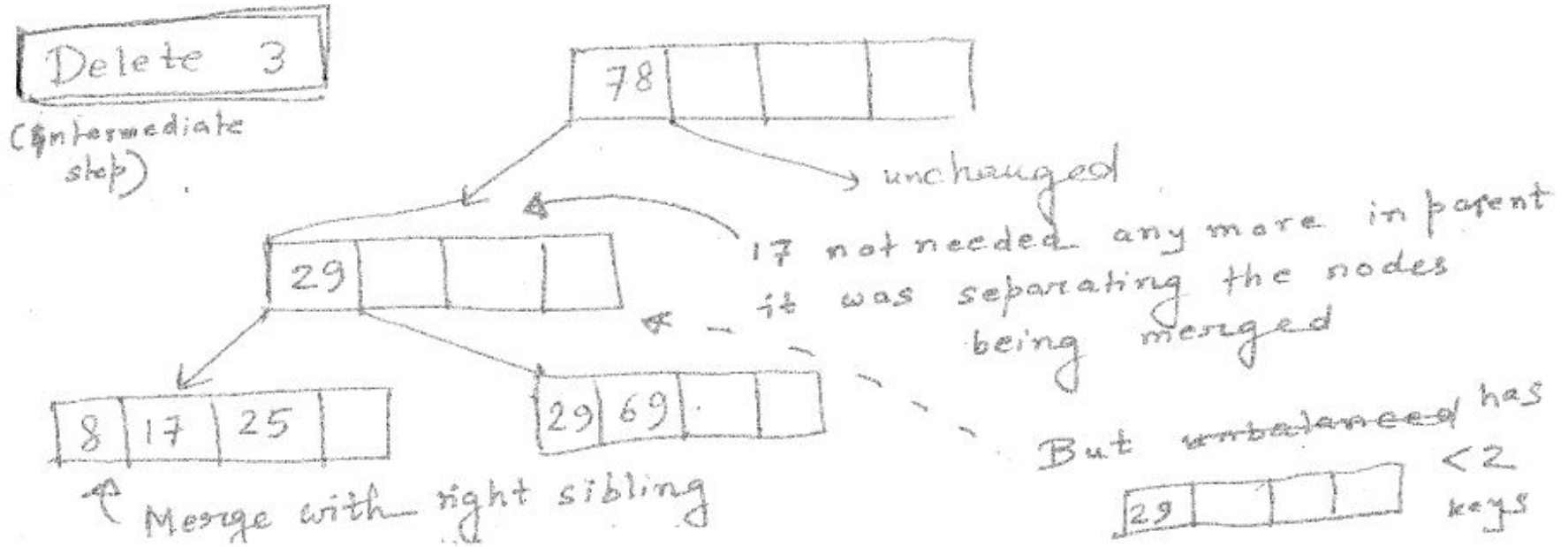
Delete 57



Note :- Next leaf on right may not be a sibling (= same parent)

Problem 1:

B+ tree insertion and deletion



Now: -

Merge

29			
----	--	--	--

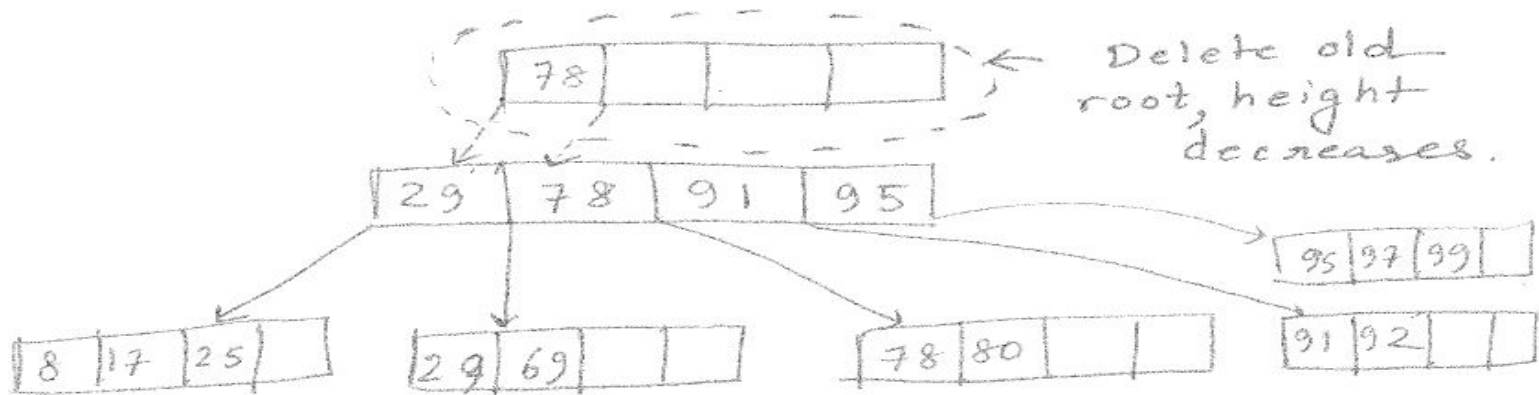
 with its sibling

Note 1: - cannot borrow from sibling

Note 2 :- Need to include key separating these siblings from parent (here 78)

Problem 1:

B+ tree insertion and deletion



Note 3: We always have space to include the key separating the siblings.

left sibling: $\leq d-1$ keys (that's why needed more keys)

right sibling: $\leq d$ keys (that's why we could not borrow from it)

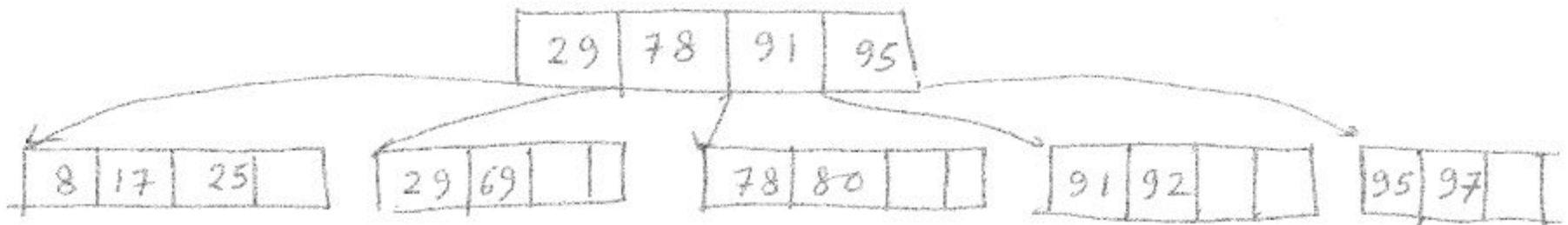
Total $\leq 2d-1$ keys.

So now we have room to include the key from parent (here 78)

Problem 1: B+ tree insertion and deletion

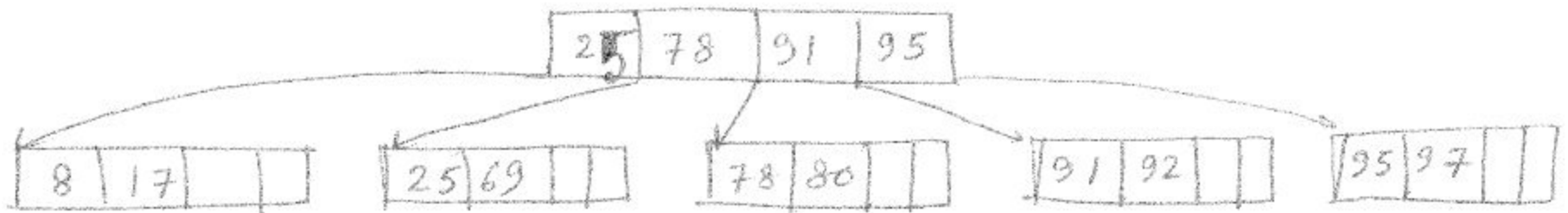
Delete 99

Easy!



Delete 29

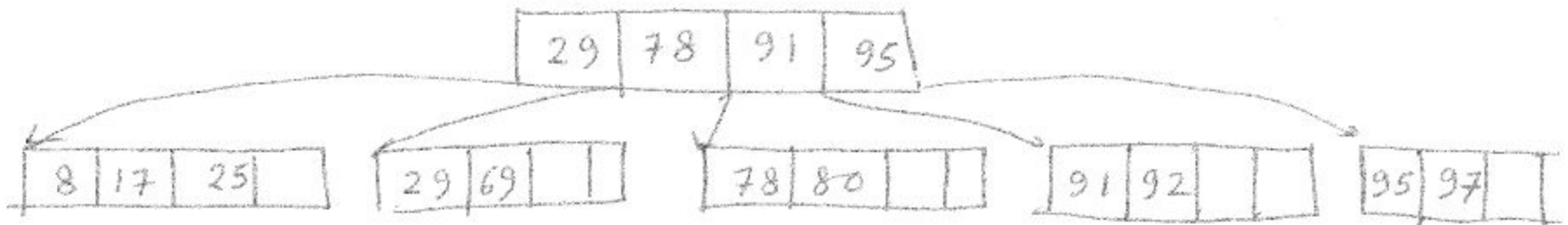
can borrow from left sibling,
update key in parent



Problem 1: B+ tree insertion and deletion

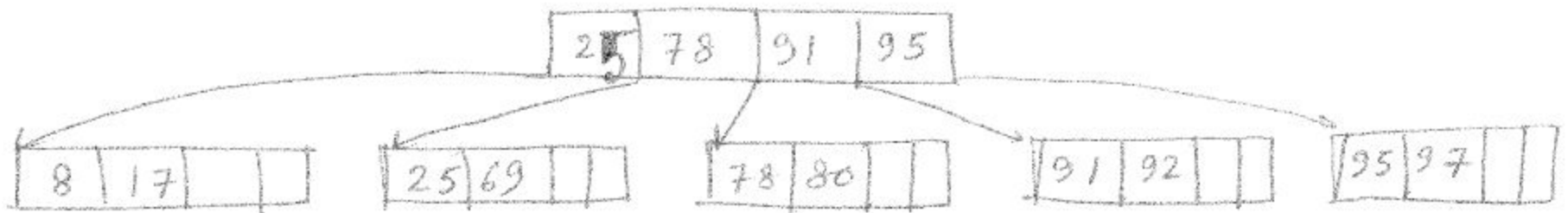
Delete 99

Easy!



Delete 29

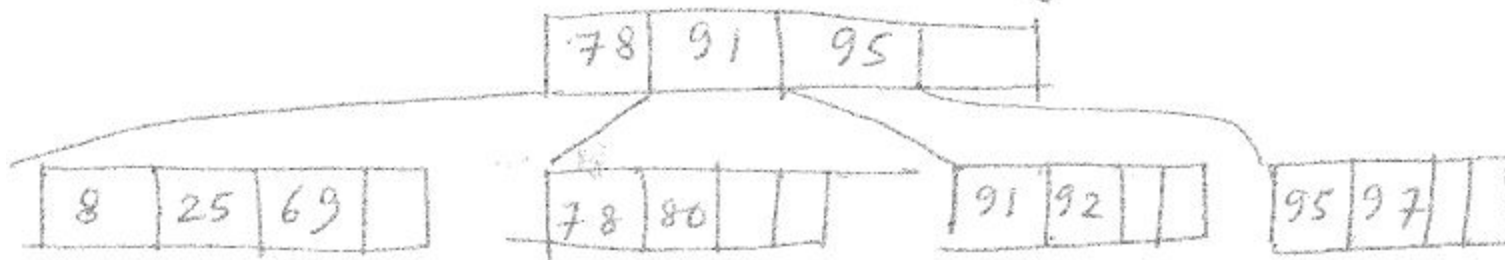
can borrow from left sibling,
update key in parent



Problem 1: B+ tree insertion and deletion

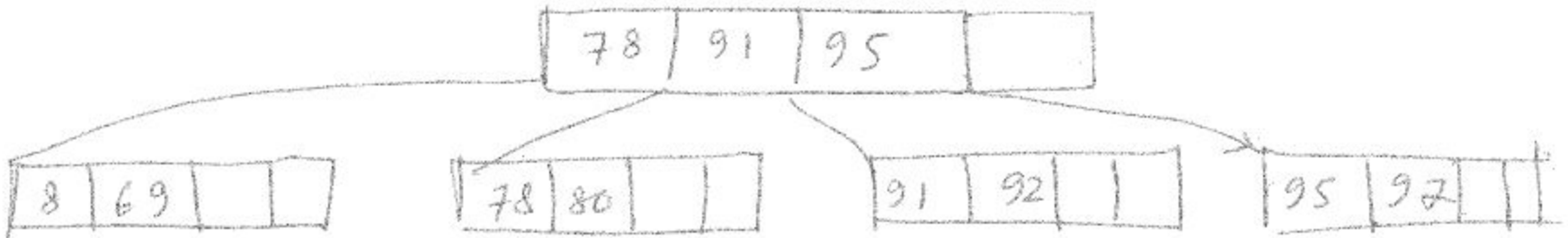
Delete 17.

can't borrow, merge with right sibling
Delete separating key (25) from parent



Delete 25

Easy!

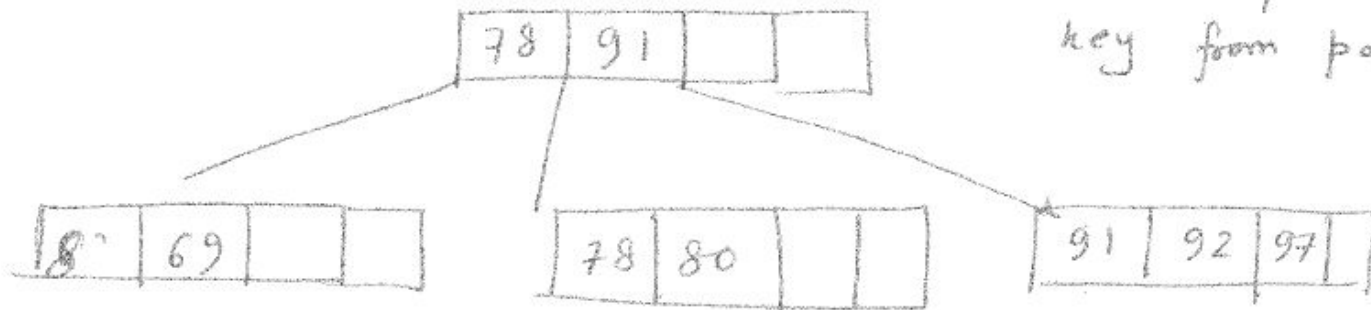


Problem 1: B+ tree insertion and deletion

Delete 95

Merge with left sibling

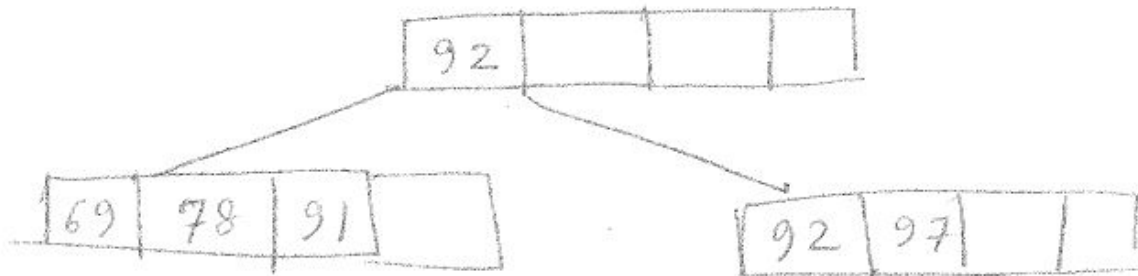
delete separating
key from parent



Delete 80, 8

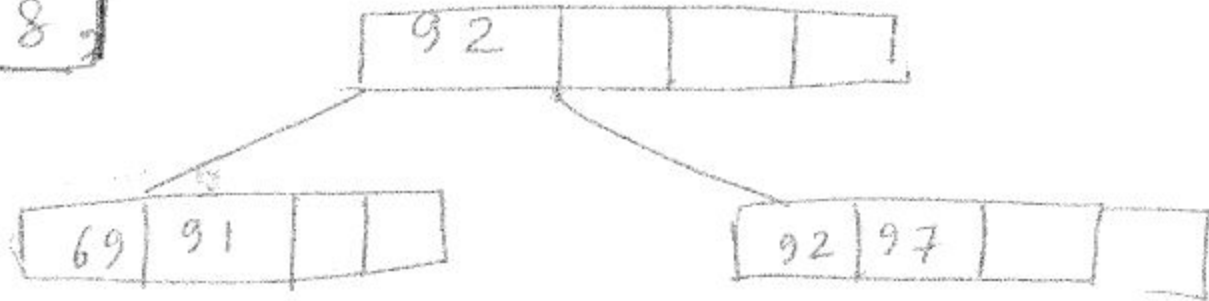
Now you can do the rest!

some steps are
skipped here.



Problem 1: B+ tree insertion and deletion

Delete 78



Delete 92

Merge leaves, root is empty, height decreases again.



Delete 69, 97, 91

Too easy!