* AVL Trees - self balancing BST

1) Insertion: LL, RR, LR (LL on A, RR on B),
RL (RR on A, LL on B)

2) Deletion:

R-TYPE	L-TYPE
R0 → LL	L0 → RR
R1 → LL	L1 → RL
R-1 → LR	L-1 → RR

3) Nodes @ height h:

$N(h) = N(h-1) + N(h-2) + 1$ $\boxed{N(0) = 1}$

Always $N(h-1) > N(h-2)$

$N(h) > 2N(h-2) + 1$

$\Rightarrow N(h) > 2N(h-2)$

$\boxed{N(h) > 2^i \times N(h-2i)}$

$\boxed{N(h) > 2^{h/2}}$ $\boxed{h \leq 2\log_2 n}$

$\boxed{h = O(\log_2 n)}$

* SPLAY TREES - BST, self adjusting

AVL - storage of height fields

SPLAY -

i) blind adjusting version of AVL

ii) Amortized time of operations : $\boxed{O(\log n)}$

iii) worst case $\boxed{O(n)}$
↓
skewed

iv) M-times traversal - $\boxed{m \times O(n)}$

Zig - LL
Zag - RR
Zig-Zig - (LL)*2

Zag-Zag - (RR)*2
Zig-Zag - LR
Zag-Zig - RL

* Amortised Analysis → Aggregate accounting potential

Aggregate
↳ stack operations, binary counter
↓

PUSH(S,x) : O(1) each
POP(S) : O(1) each.
Multipop : min(s,x)
↳ worst case - O(n)
Has n operations
worst case cost of sequence is $O(n^2)$

O(nk)
cost of increment
= no. of bits flipped.

$$\text{No. of flips} = \sum_{i=0}^{k-1} \text{floor}(n/2^i)$$

$$= n \sum_{i=0}^{\infty} \text{floor}\left(\frac{1}{2^i}\right) = 2n$$

n increments cost O(n)
Avg cost = O(1)

AMORTIZED ANALYSIS

↳ Analyse a sequence of operations on a data structure

* GOAL — show that although some individual operations may be expensive, on avg the cost per operation is small.

* METHODS — Aggregate, Accounting, potential.
  |
  stack operations, binary counter.

• stack operations;

1. PUSH (S, x) : O(1) each ⇒ O(n) for any seq of n operations

2. POP(S) : O(1) each ⇒ O(n) " " "

MULTIPOP(S, k)

while S is not empty and k > 0

do POP(S)

k ← k - 1.

↳ Multipop : # of iterations of while loop is min(s, k), where s = # of objects on stack.

∴ total cost = min(s, k)

3. Sequence of n push, pop, Multipop operations;

- worst-case cost of MULTIPOP is O(n)
- Has n operations
- Worse case cost of sequence is $O(n^2)$

Observations :

each object can be popped only once per time

if there are $\leq n$ pushes $\Rightarrow$ $\leq n$ pops, including those in multipop.

$\therefore$ Total cost = $O(n)$

Average over the n operations = $O(1)$ per operation on average.

* Binary-counter :

- k-bit binary counter $A[0 \ldots k-1]$ of bits, where $A[0]$ is least significant bit, $A[k-1]$ is most sig.
- counts upward from 0
- initially, counter value is 0, so $A[0 \ldots k-1] = 0$.
- To increment,

```
INCREMENT (A, k)
    i ← 0
    while i < k and A[i] == 1
        do A[i] ← 0
           i ← i + 1
    if i < k
        then A[i] ← 1
```

- example:

$k = 3$

counter value	2	1	0	cost
0	0	0	0	0
1	0	0	1	1
2	0	1	0	3
3	0	1	1	4
4	1	0	0	7
5	1	0	1	8
6	1	1	0	10
7	1	1	1	11
0	0	0	0	14

∴ cost of increment = O(# of bits flipped)

each call could flip k bits, so: n INCREMENTS takes $O(nk)$ time.

- observation:

not every bit flips every time

bit	flips how often	times in N INCREMENTS.
0	every time	n
1	1/2 the time	$n/2$
2	1/4 time	$n/4$
⋮		
i	$1/2^i$ the time	$(n/2^i)$

- Total no: of flips $= \sum_{i=0}^{k-1} \text{floor}(n/2^i)$

$$= n \sum_{i=0}^{\infty} \text{floor}\left(\frac{1}{2^i}\right)$$

$$= 2n.$$

∴ n increments costs $O(n)$

Avg cost per operation = $O(1)$

* ACCOUNTING METHOD

- Assign different charges to diff operations
  - some are charged more than actual cost
  - some are charged less
- Amortized cost = amount we charge.
- when amortized cost > actual cost, store the difference on specific objects in the data structure as credit.
- use credit later to pay for operations where actual cost > amortized cost.

DIFFERENCE → in accounting method, diff operations can have diff cost
in aggregate analysis, all operations have same cost.

Let c_i = actual cost of ith operation

$\hat{c}_i$ = amortized cost of ith operation

for all sequence of n operations,

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

Total credit stored = $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$

* STACK operations

operation	actual cost	amortized cost
PUSH	1	2
POP	1	0
Multipop	Min(k, s)	0

when pushing an object, pay $2

- $1 pays for PUSH
- $1 - prepayment for POP, MULTIPOP

since each object has $1, which is credit, the credit can never go -ve.

∴ total amortised cost i.e O(n) is an upper bound on total actual cost

* Binary counter

- charge $2 to set a bit to 1

  $1 – pays for setting a bit

  $1 – prepayment for flipping it back to 0

- Have $1 credit for every 1 in the counter.

  ∴ credit ≥ 0.

* Amortized cost of increment

cost of resetting bits to 0 is paid by credit

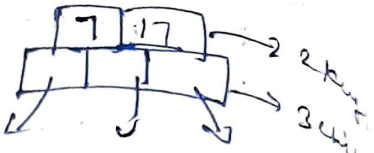
At most 1 bit is set to 1

therefore, amortized cost ≤ $2

for n operations, amortized cost = O(n)

14/

* Multiway search tree



1. Generalization of BST

2. No: of keys per node;

- 1 (BST)
- > 1 (m-way)

3. Max no: of children possible in ordering the tree

M-way search tree:

m-way search tree may be empty.

If its not empty, it satisfies;

1. each node has m children (order of tree)

↓

(atmost)

2. If a node has k child nodes ($k \leq m$), then the node has exactly (k-1) keys.

↓

data / value stored in BST / m-way.

3. keys in each node are in ascending order

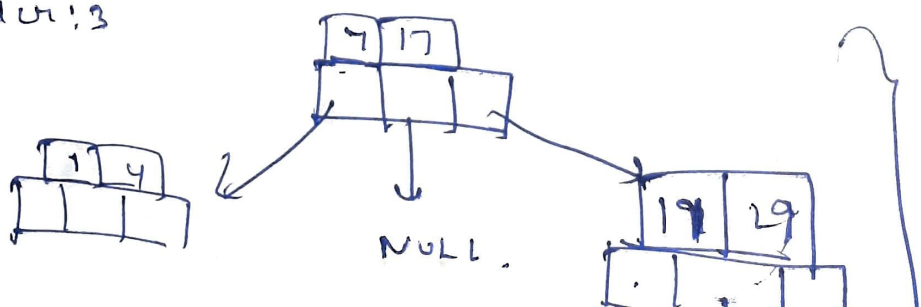
4. Left child value < parent value

Right " " > parent value.

↓

keys in Left / right subtree of key i' are smaller than i.

order: 3



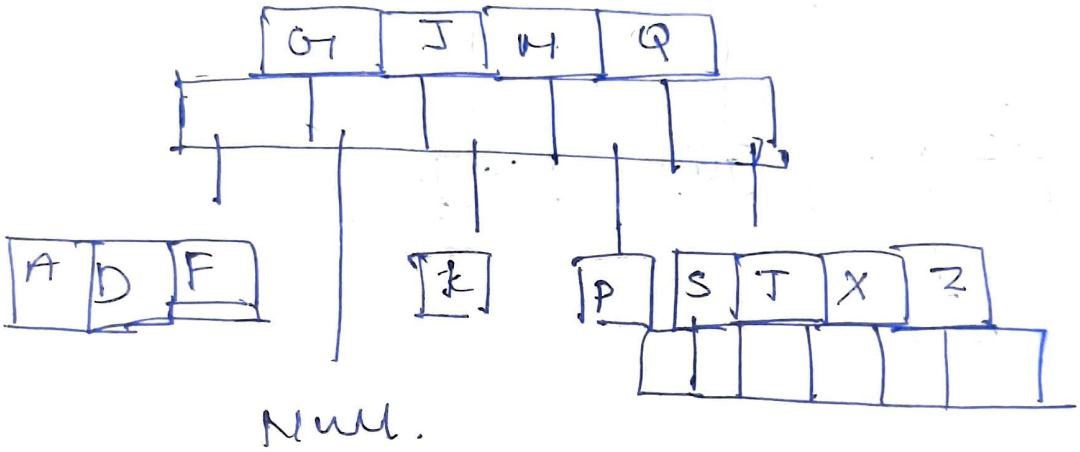
example

* Deletion in m-way search tree

a b c d e f g h i j k l m n o p q r s t u v w x y z

- Insert in 5 way search tree

G, J, M, Q, P, F, D, A, K, T, X, Z, S.



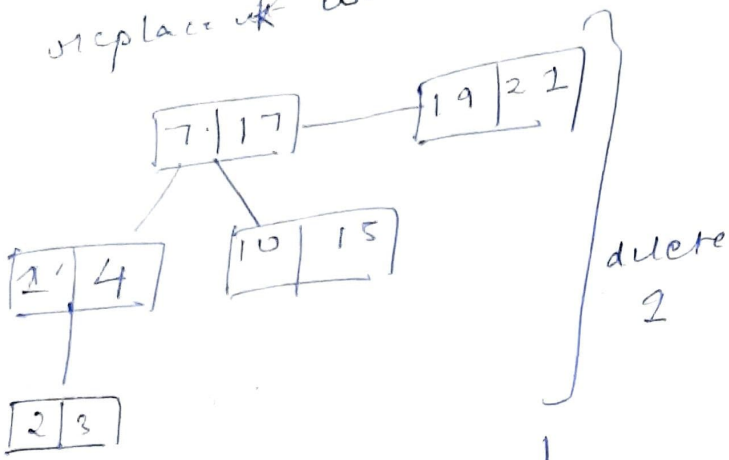
Deletion:

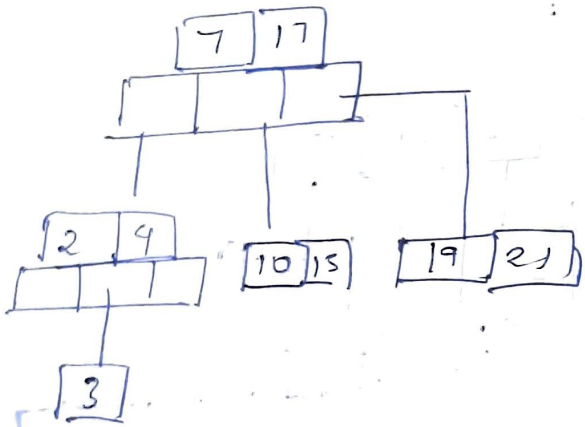
case 1:

If left + right are NULL, dont replace anything, just delete it.

case 2.

Left subtree of K is NULL
& right is not NULL;
choose smallest key K' from right subtree
replace K with K'

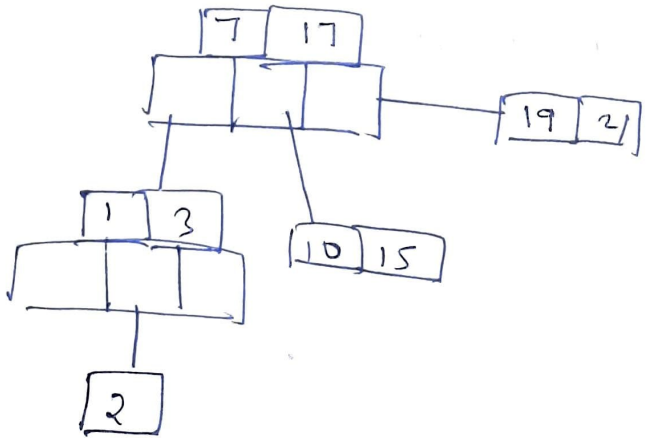




case - 3

Right of K NULL
Left of K Not Null,
→ Largest of left subtree replaces K.

Delete 4 (Return top for tree B4 deletion)

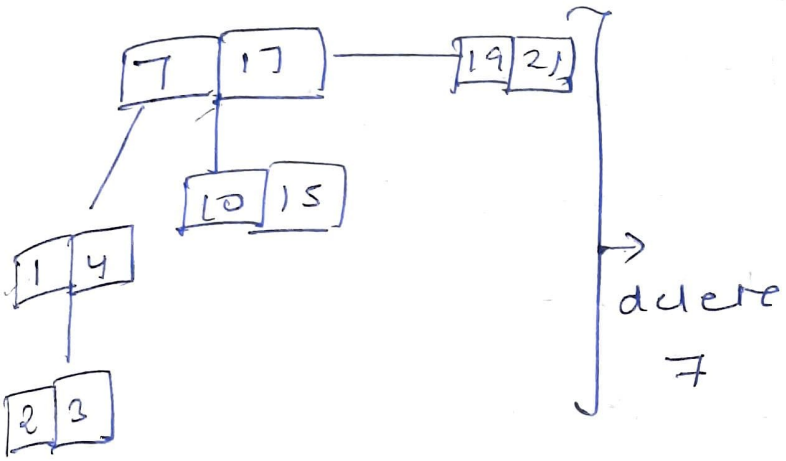


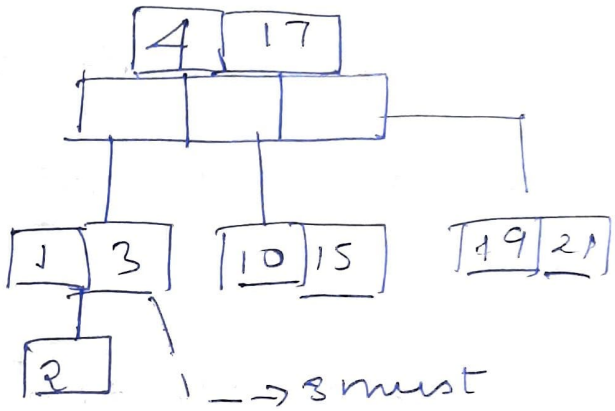
case 4

- both are non-empty.

Thus, we can either replace k with

- 1. smallest of right subtree (k') [or]
2. highest of left subtree (k'')

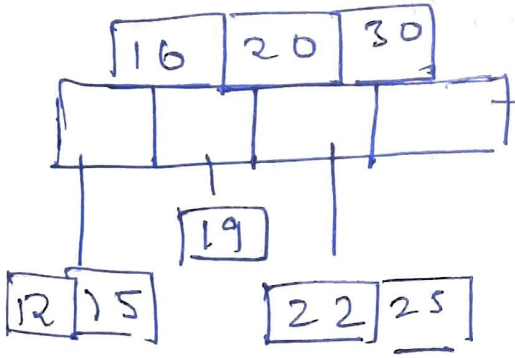


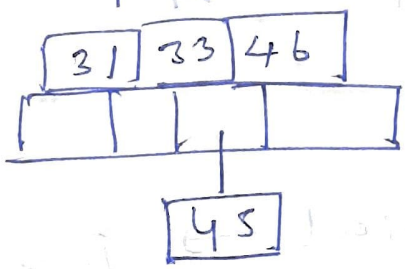


→ 3 must replace 4's place (recursive deletion).

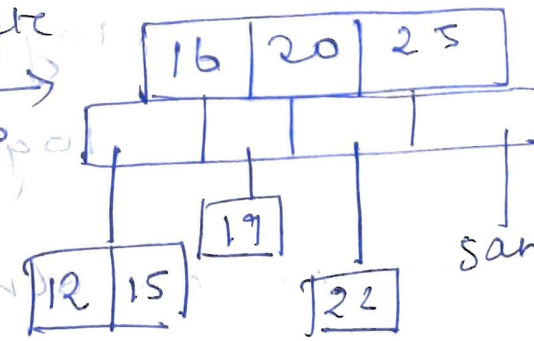
---

delete 30, 46, 25, 19, 45, 20

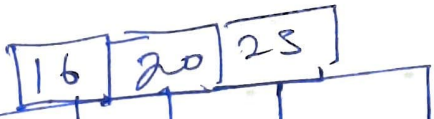




delete 30



delete 46.



* Analysis of M-way search

- complexity of any operation is $O(h)$
- Min number of elements : h
- Max no: of elements :

At height 1 – $m-1$ keys

2 – $m(m-1)$ keys

At height h : $m^{h-1}(m-1)$ keys

Total no: of elements : $m-1 + m(m-1) + \cdots$

$$m^{h-1}(m-1)$$

$$= (m-1)(1+m+m^2+\cdots m^{h-1})$$

$$= (m-1)(m^h-1)/m-1 = \boxed{m^h-1}$$

- If a m-way search tree has n elements, height h varies from a min of n to a max of m^h-1
- Best case: $n = m^h-1$

$$\log_m m^h = (h-1)\log_m m$$

$$\log_m n+1 = h$$

- worst case : $O(h)$
- Balancing is essential → B trees.

B-Trees

↳ A B-Tree of order m is a m-way search tree & hence may be empty.

If non-empty, it satisfies the following:

1. root node must have at least 2 children, atmost m

2. internal nodes other than root node must have atleast $\lceil \frac{m}{2} \rceil$ non-empty child nodes, at most m non-empty children

3. All external nodes must be at same lvl

4. no: of keys is always 1 less than no: of children

ex: B tree order = 3

↓

implies root should have min 2 child to max of 3,

other non-leaf nodes will have 2 - 3,

order = 5 :

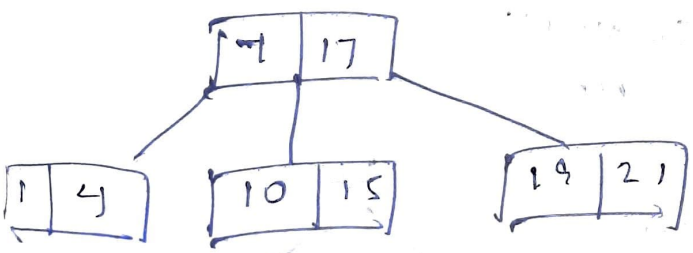
root (min 2 to 5)

non-leaf : [min 3 to max 5]

↳ $\lceil \frac{5}{2} \rceil = \lceil 2.5 \rceil = 3$

↓

ceiling function.



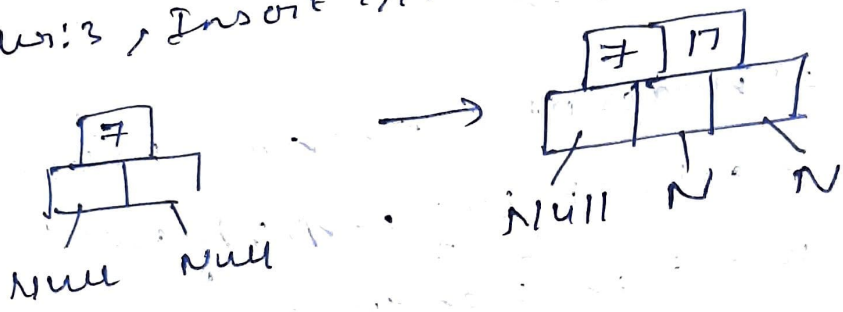
↳ example of B-Tree.

* B-Tree Insertion

case: 1

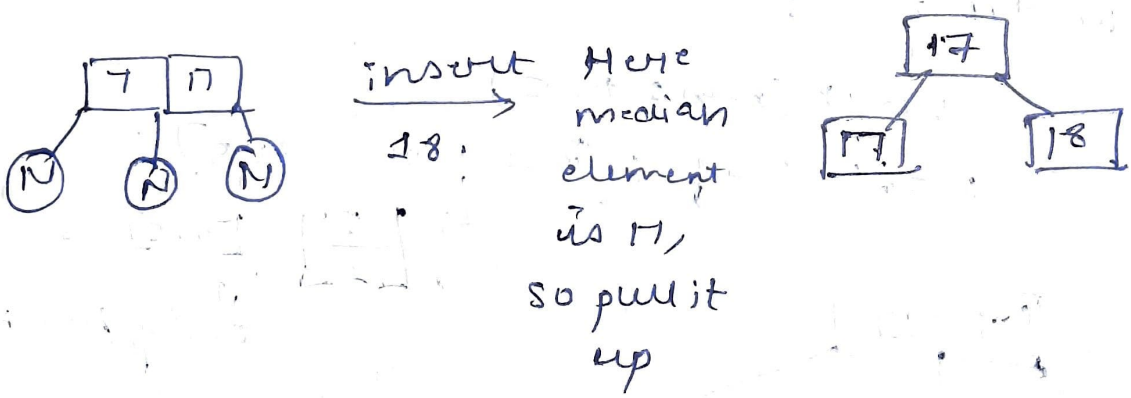
1. If node can accomodate k, then insert it in the node & adjust pointers

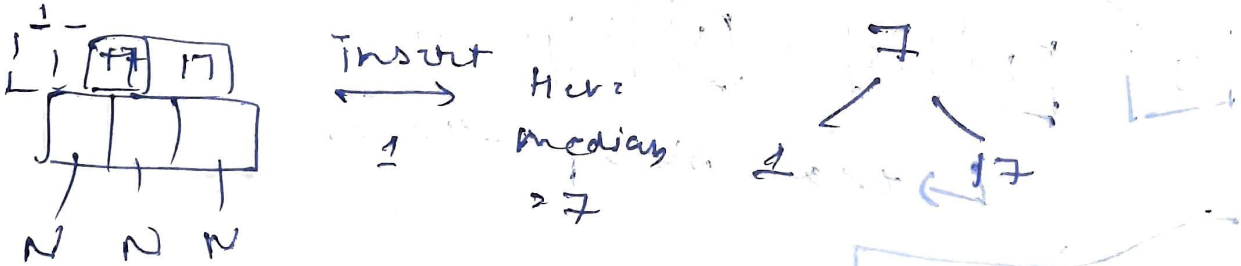
order: 3, Insert 7, 17



case 2

If node cant accomodate k, then logically insert the k in approp. position & split the list into 2 ~~nodes~~ at its median k median k median is pulled up & inserted in parent node. The keys less than k median form the left child of k median and keys greater than k median form right child of k median.





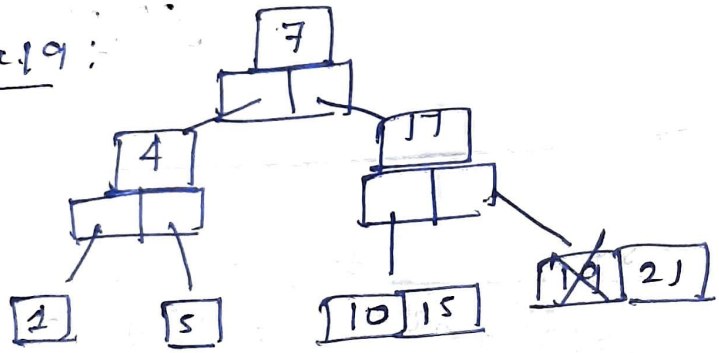
* Deletion for B-Tree

1. deletion in leaf node } doesn't violate rules.
2. Non-leaf node
3. } opposite
4.

case 1:

If key is in leaf node & deletion, doesn't leave node less than min no: of elements, then just delete the key with its appropriate right

delete 19:



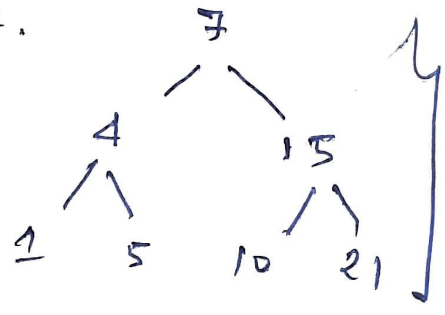
} doesn't violate rules.

case 2:

if key to be deleted is in non-leaf node, if deletion of key satisfies the min rules, then replace deleted with largest key of left subtree / smallest of right subtree

- may trigger case 1

Delete 17.



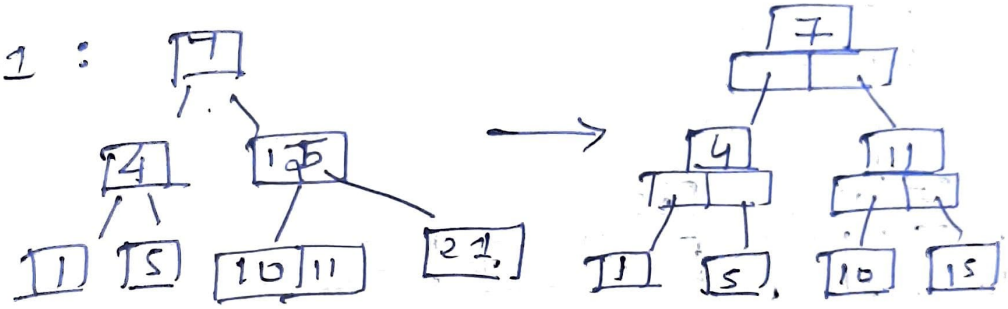
} don't replace 17 with 21 because leaf node will become null.

case 3 :

if deletion of key (k) violates the minimum rules, elements are borrowed from one of its left / right sibling nodes.

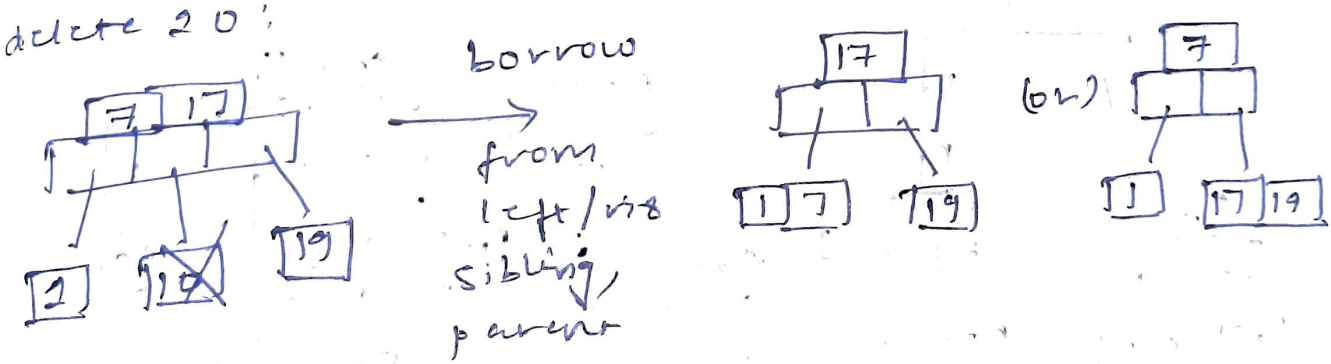
Thus, if left sibling has keys to spare, move largest key from left subtree to the parent node. The intervening element in the parent node is moved down to set the vacancy created by k.

delete 21 :



case 4 :

delete 20 :



If deletion of key k leaves node with less than its min no: of elements and if both siblings dont have elements to spare, node is merged with one of sibling nodes along with intervening element in the parent node.

* Analysis of B-Trees

B-Tree is m-way search tree.

complexity is O(h)

upperbound: B tree of order m, height h.

with n elements satisifies $n \leq m^h - 1$

lowerbound:

min no: of nodes at lvl 1 : 1

lvl 2 : 2

3 : $2 \cdot \lceil \frac{m}{2} \rceil$ (ceiling function)

4 : $2 \cdot \lceil \frac{m}{2} \rceil^2$

$$\text{or} \begin{cases} h \text{ th} : 2 \cdot \lceil \frac{m}{2} \rceil^{h-2} \\ h+1 : 2 \cdot \lceil \frac{m}{2} \rceil^{h-1} \end{cases}$$

Number of elements $n \geq 2 \cdot \lceil \frac{m}{2} \rceil^{h-1} - 1$

$$\log_m (n+1) \leq h \leq \left\lfloor \log_{\lceil \frac{m}{2} \rceil} \frac{(n+1)}{2} \right\rfloor + 1.$$

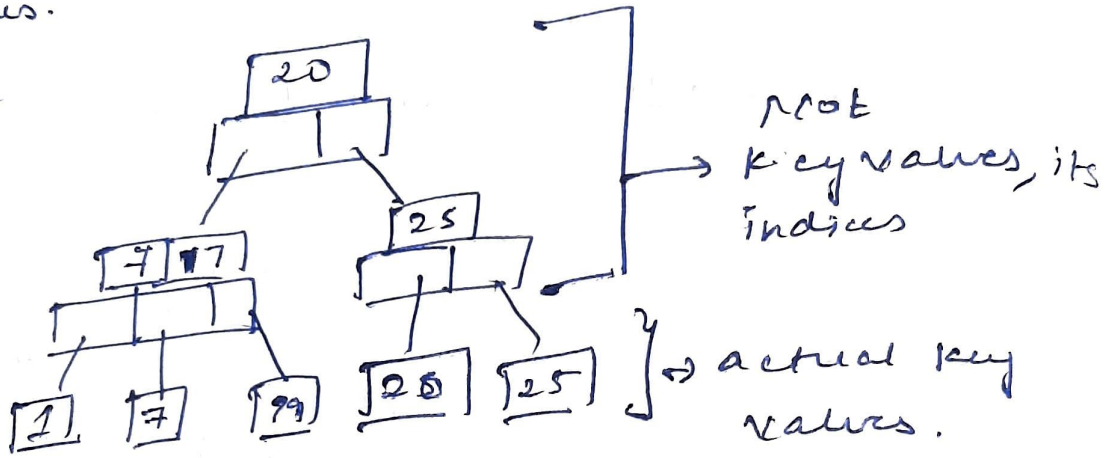
2/9/20

* B+ Trees

similar to B trees with keys stored only in leaf nodes.

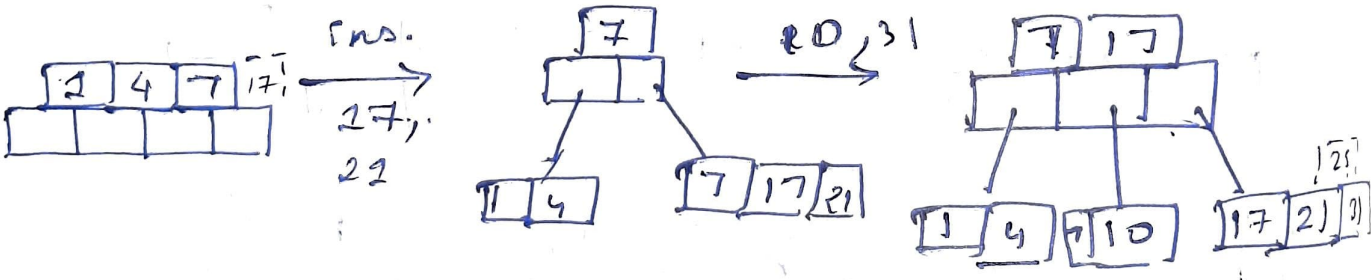
All internal nodes are indices to the keys in leaf nodes.

example:

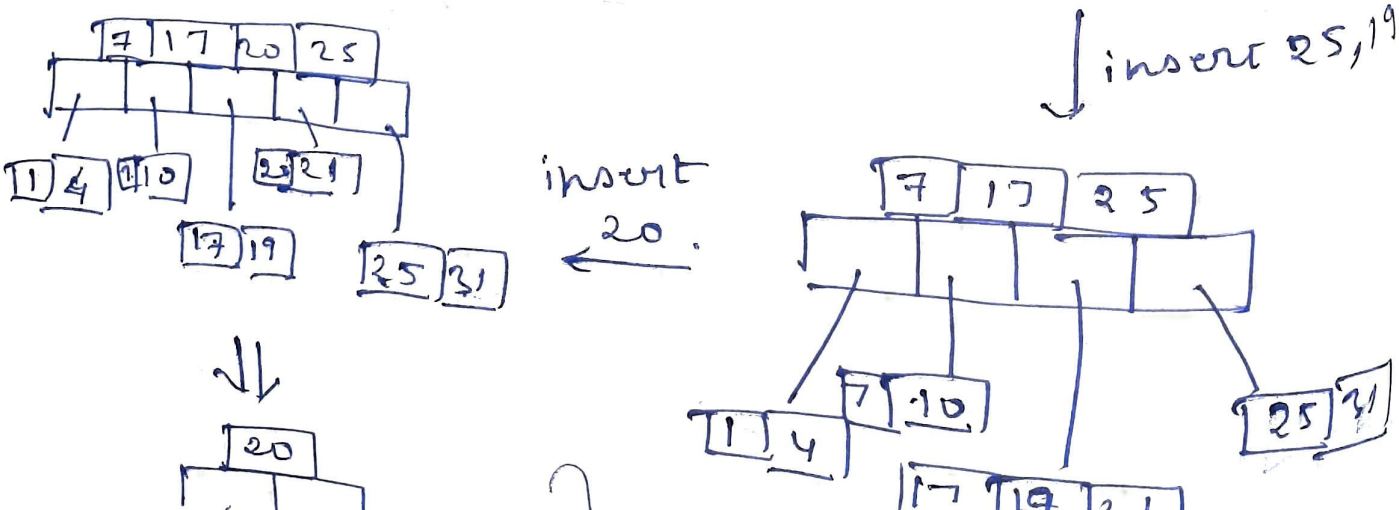


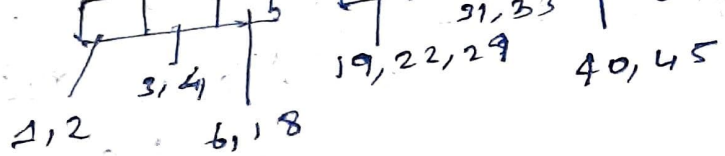
* B+ Trees insert (order=4)

1, 4, 7, 10, 17, 21, 31, 25, 19, 20.



insert 25, 19





* Deletion of B+ Trees

case 1: If its a leaf node, then just delete key

case 2: non-leaf node satisifying the minimal rule, then replace it with left/right extreme of its R/L subtree

case 3: if it is not satisifying minimal rule, then borrow key from left sibling if possible.

case 4: If left sibling not possible to borrow, merge with any 1 of the siblings along with intervening parent.

example: order = 5