# MODULE-2

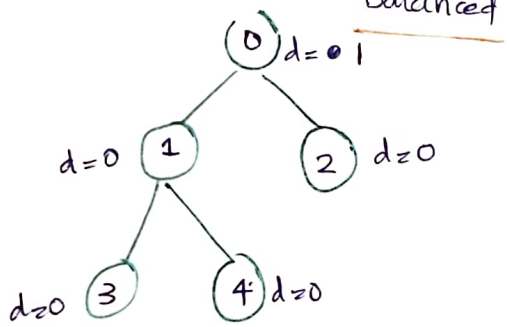
## BALANCED BINARY SEARCH TREE

→ It is type of binary tree in which the difference between the height of the left and right subtree for each node is either 0 or 1.

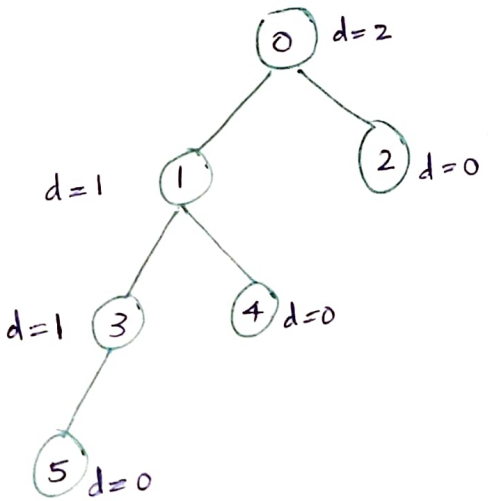
→ A single node is also balanced. It is also reffered to as a height balanced binary tree.

→ An Empty tree (Root = NULL) is also always considered as balanced.

Balanced




Unbalanced.




Depth of a node = height of Left - height of right.

D = HL - HR

To check whether a tree is balanced conditions:

① • The absolute difference b/w heights of left and right subtrees at any node should be less than 1.

$$|H_L - H_R| <= 1$$

② • for each node the left subtree should be a balanced binary tree.

③ • for each node, its right subtree should be a balanced binary tree.

## AVL TREE

→ An AVL tree is a self balancing Binary search tree (BST) where the difference between the heights of left & right subtrees of any node cannot be more than one.

→ It follows the general properties of a Binary search tree.

→ Each subtree of a tree is balanced ie, The difference between the height of the left and right subtree is at most 1.

→ The tree balances itself when a new node is added. Therefore the insertion operation is time consuming.

## Advantages:

→ AVL trees can self balance
→ It also provides faster search op.
→ AVL trees also have balancing capabilities with a different type of rotation.

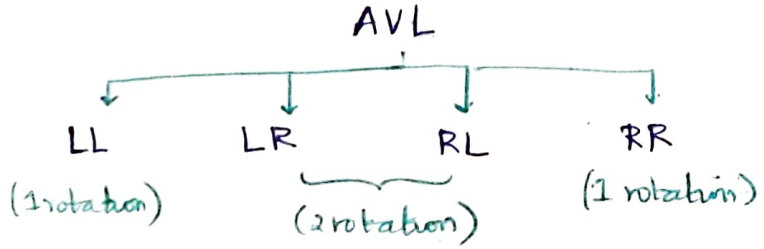
## Disadvantages:

→ AVL trees are difficult to implement
→ AVL trees have high constant factors for some operations.

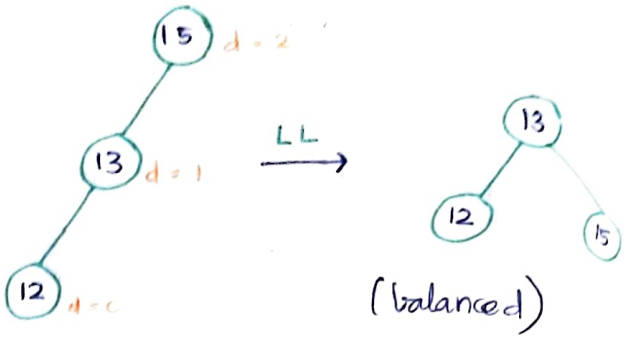
→ If a tree 'T' is a non empty binary tree in which ($T_L$ & $T_R$) are the left and right of Tree T.
The Tree is an AVL tree if and only if

1) $|H_L - H_R| <= 1$
2) $T_L$ is an AVL tree
3) $T_R$ is an AVL tree.

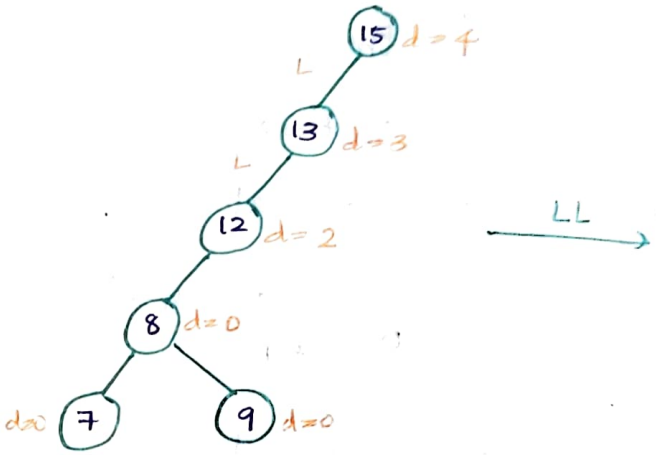
→ It is balancing itself through performing rotation.

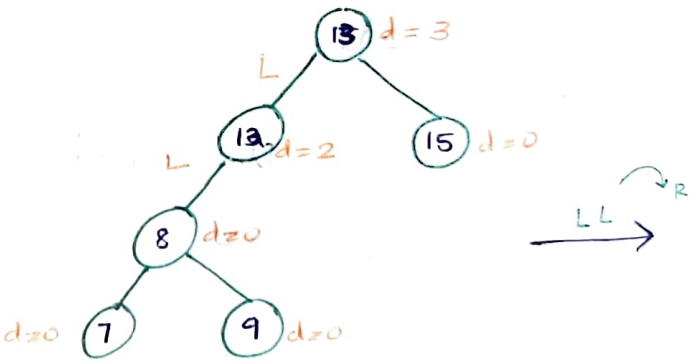


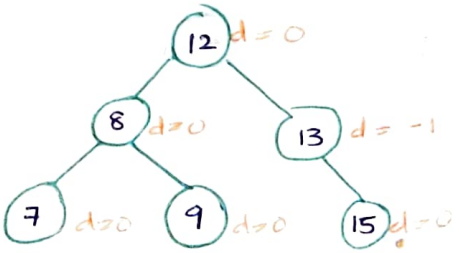

Eg: 1) 15, 13, 12




2) 15, 13, 12, 8, 7, 9



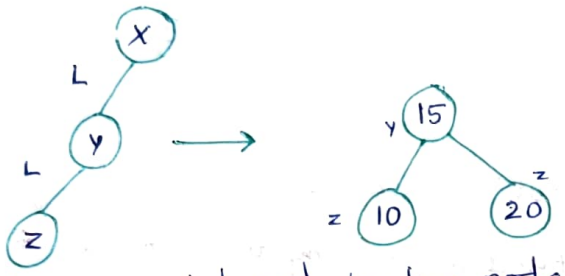



3) ~~15, 13, 12, 20, 35, 14~~

Case I

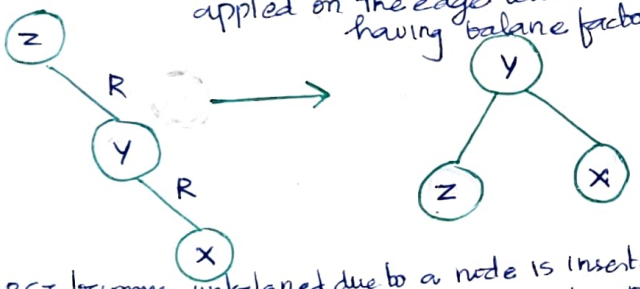
x, y, z  x = 20, y = 15, z = 10




when BST becomes unbalanced due to a node inserted into the left subtree of the left subtree of x then we perform LL rotation.

LL → clockwise rotation which is appled on the edge below a node having balance factor 2.

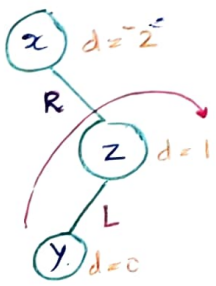
Case II




when BST becomes unbalanced due to a node is inserted to right subtree of the right subtree of z then we perform RR Rotation.

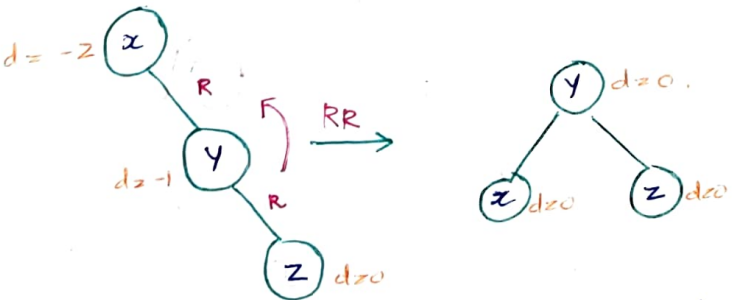
CASE III  x, z, y (consider x > y > z)

RR → anticlockwise applied on the edge below a node having balance factor -2 or 4bbx.




RL rotation

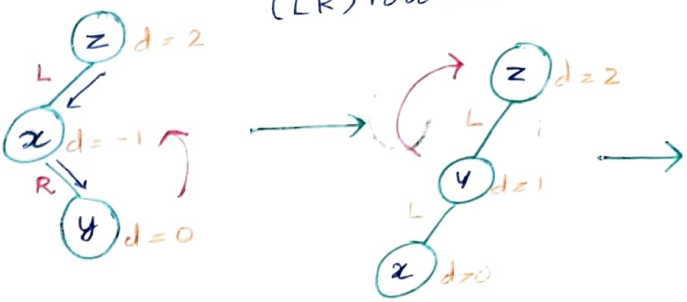
(as the first diversion ist to R we must convert it to case RR).

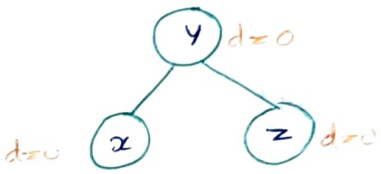



RL = LL+RR. first LL rotation is peeformed on subtree of then RR rotation is performed on full tree. by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, and 1 (same at LR) stuff in RR+LL

Case IV  z, x, y.

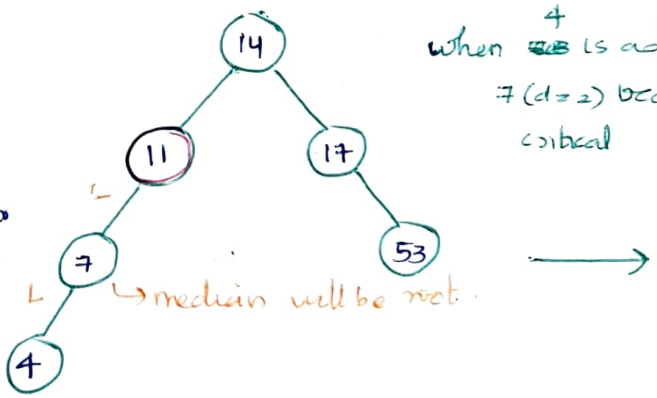
(LR) rotation.



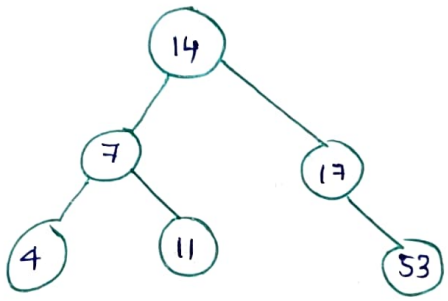




8) Construct an AVL tree by inserting the following data

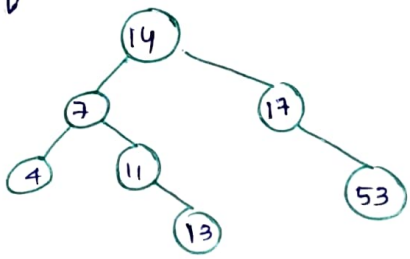
14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20.

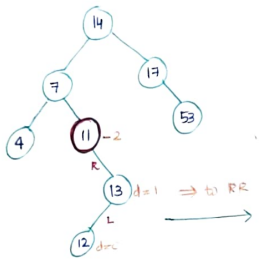


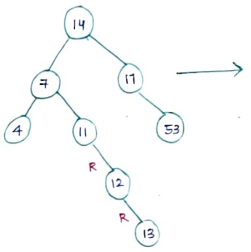

when 4 is added 7 (d = 2) became critical

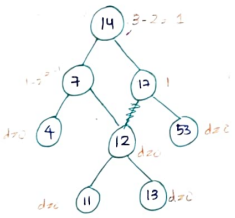



Inserting after 4, ie 13.

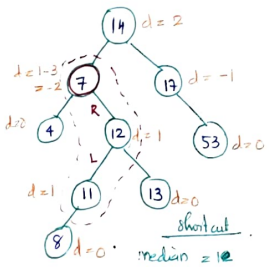








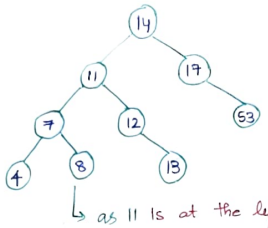
Then adding elements from 8.



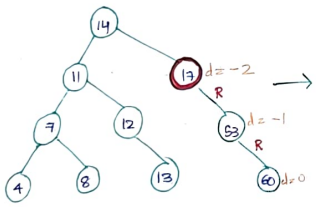
median = 12

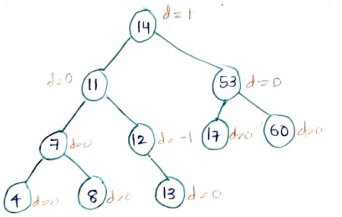
7, 11, 12

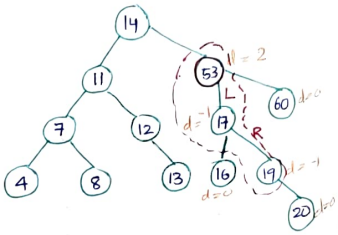
11 → Root 7 & 12 childs



↳ as 11 is at the left of 11 but 7 is already at the left of 11 so then 7 and 8 are compared and as 8 is larger it is placed at the right of 7.

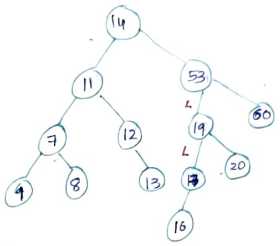


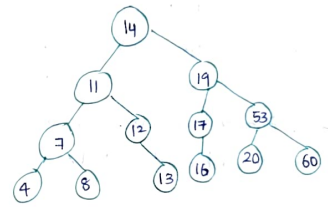




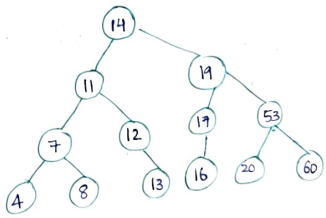


Shortcut = 17, 19, 53





## DELETION IN AVL TREE

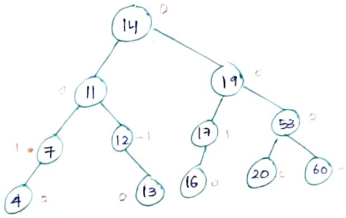


Delete 8, 7, 11, 14, and 17 from the BST or AVL tree.

→ Searching the element 8 is less than 14 move to 11 (11 > 8) move to 7 (7 < 8) move to right we found 8.

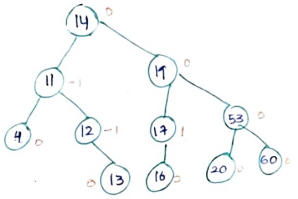
after deletion →

after deletion we need to check the balance factor only after that we will delete the other element.



after Deleting 7

14 → 11 → 7 (one child).



after deleting 11

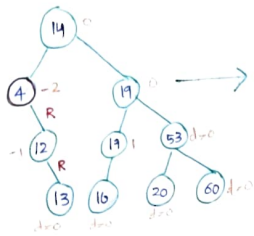
14 → 11

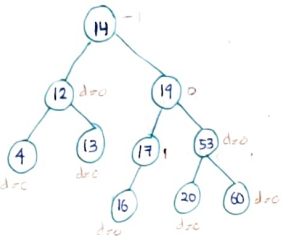
11 → is having 2 children.

replace the child with Inorder predecessor or inorder successor.

Inorder predecessor → largest element of left

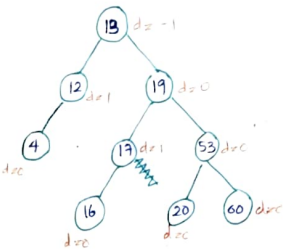
Inorder successor → smallest element from the right.





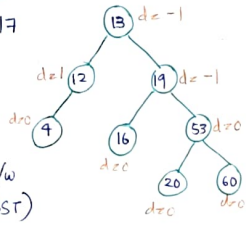
after deleting 14

14 is the root and having 2 children



after deleting 17

13 → 19 → 17



(Refer the diff b/w AVL & BST)

# RED-BLACK TREES.

(Imp points)

- Red black tree is a binary search tree.

→ The prerequisite of the red-black tree is that we should know about the binary search tree.

→ Each node in Redblack tree contains an extra bit that represents a color to ensure that the tree is balanced. during any operations performed on the tree like insertion, deletion etc...

→ In Binary search tree, the searching, insertion and deletion takes

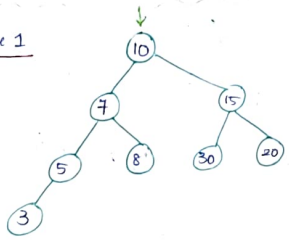
$O(\log_2 n)$ → average case

$O(1)$ → Best case

$O(n)$ → worst case

different scenarios of Binary search trees.

case 1

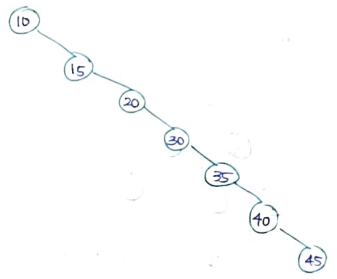


If we want to search so we compare it with root node. as it is greater it is then compared with right subtree. and this goes on. and atlast we reach the root node and it will show the element is not found on the tree.

→ After each operation the search is divided into half.

The above BST will take $O(\log n)$ time to search element.

case 2



The above tree shows a right skewed binary tree. so if we need to find so we search and compare each element so the right skewed BST will take $O(N)$ time to search an element.

→ one case the BST is balanced and other case it is unbalanced.

∴ A balanced tree takes less time to perform an operation than an unbalanced tree.

- Since AVL tree is a height balanced tree. Red-Black trees are used because the AVL trees requires many rotations when the tree is large, whereas R-B tree requires a maximum of 2 rotations to balance the tree.

→ difference between AVL and redblack tree is that AVL is strictly height balanced but Redblack tree is not completely height balanced.

∴ AVL is more balanced than Redblack.

→ Redblack tree guarantees $O(\log_2 n)$ time for all operations.

→ Insertion is easier in AVL and deletion and searching are easier in Redblack and as it require on fewer rotations.

→ As the name suggests that the node is either colored red/black and sometimes no rotation is needed only recolouring is required to balance the tree.

## Properties

Red-black (roughly)

→ It is a self balancing tree. It balances the tree itself by rotations or recolouring the nodes.

→ each node is in Red or black colour. and contains an extra info knows as a bit that represents the colour of the node.

0 → black 1 → red.

pbs (imp)

→ Root node is always black.

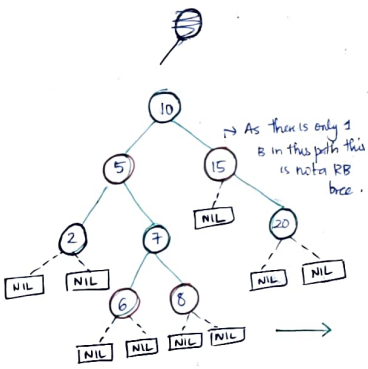
→ The nodes having no children considered the internal nodes and these nodes are connected to the nil nodes that are always in black colour.

imp → newly created node will be red color.

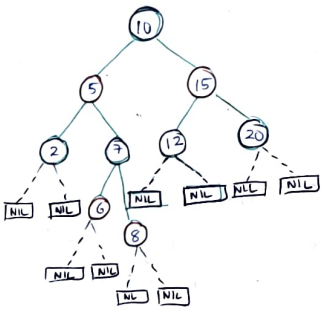
→ NIL nodes are leaf nodes in Red-Black tree.

→ If the node is red then their children should be in black colour.
ie there should be no red-red parent-child relationship.

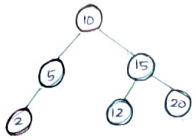
→ Every path from a node to any of its descendants nil node should have same number of black nodes.



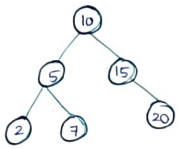

→ As there is only 1 B in this path this is not a RB tree.



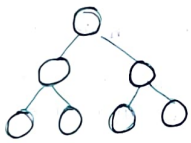

Eg:



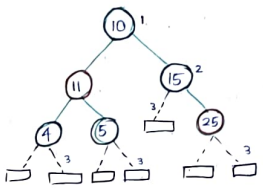

⟹ Since the root node is red and the no of black nodes in each path is not same. (not RB)




Since every path does not have the same no of black nodes.
∴ If there was another left child for 15 it would be a RB tree.



→ Every perfect binary tree containing only black nodes is also a red black tree.




→ all path have 3 black nodes

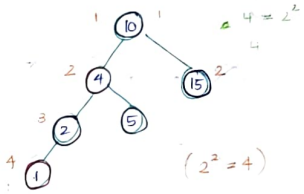
⟹ root node is black

⟹ If the node is red their children is black

But it is not Redblack tree because 11 > 10 but 11 is in the left of 10 since it is not a binary search tree it is not redblack tree.

⟹ The longest path from the root node is not more than the twice of the shortest path

ie

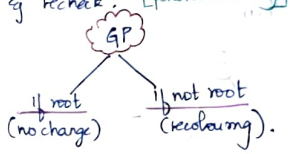



- If we add black anywhere it will be invalid, we can add red. but no more black. /also red node cannot be inserted after red nodes.

## Insertion In Red-Black Tree.

### Rules.

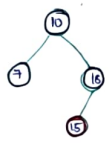
1. If tree is Empty, create newnode as rootnode with colour black
2. If tree is not Empty create newnode as leafnode with colour Red.
3. If parent of newnode is black then exit
4. If parent of new node is Red then check the colour of parent's sibling of newnode
   a) If colour is black or null then do suitable rotation & recolour. [parent + GP]
   b) If colour is red then recolour & also check if its grandparent of node is root node then recolour it & recheck. [parent + sibling]



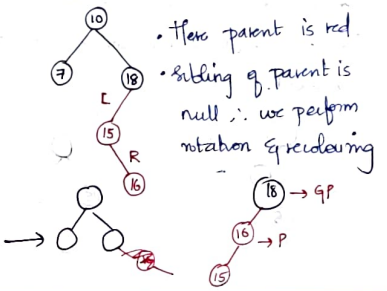

Example : 01

Insert : 10, 18, 7, 15, 16, 30, 25, 40, 60, 2, 1, 70.

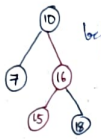

After recolouring.



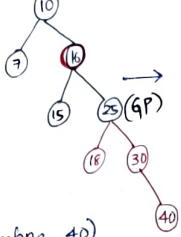

Inserting 16

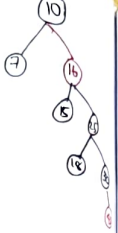



- Here parent is red
- Sibling of parent is null ∴ we perform rotation & recolouring

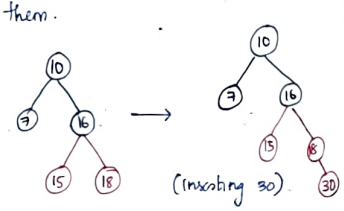



before recolouring



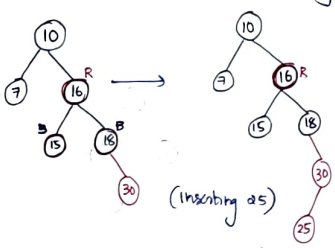




After recolouring → So since GP is 18 and P = 16 we must recolour them.



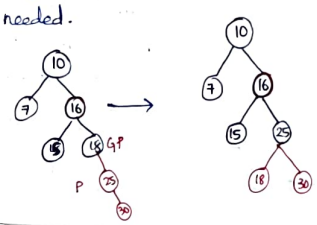

(Inserting 30)

→ Since parent is red and sibling is red (recolouring only is required) we must recolour, parent and sibling

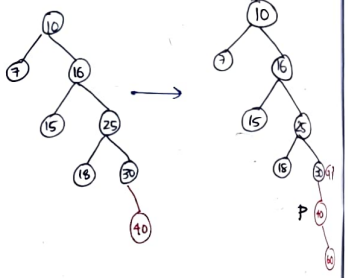



(Inserting 25)

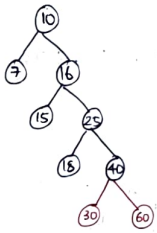
→ parent of newnode is red and sibling is null ∴ rotation and recolouring is needed.



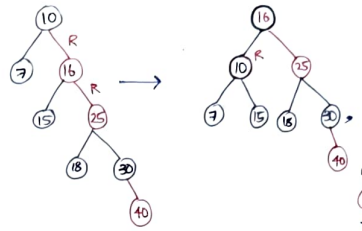

X ⇒ Since parent and sibling are red only recolouring is needed. Sibling and parent must be recoloured. (Inserting 40)



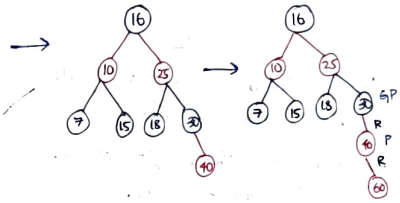

X ⇒ parent is red and sibling is null Rotation and recolouring is required.



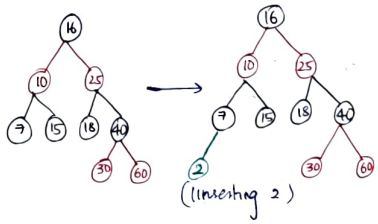

✓ ⇒ Since parent and sibling is in red colour we must recolour parent and sibling and check the GP — if root no change
— else recolour it



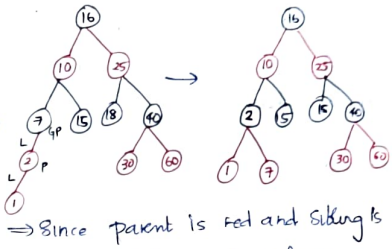

Here we consider (35) as the new node so P = 16 and GP = 10. after performing the rotation we must recolour the parent and grandparent since parents sibling is black ★



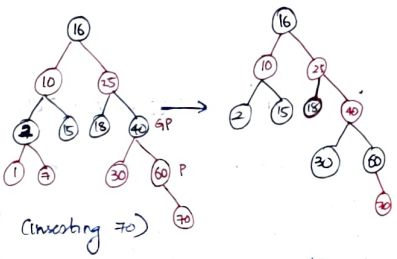

→ Since it is R-R conflict parent is red and sibling is null then rotation and recolouring is needed. [parent and GP]




⇒ If parent of new node is black Exit.

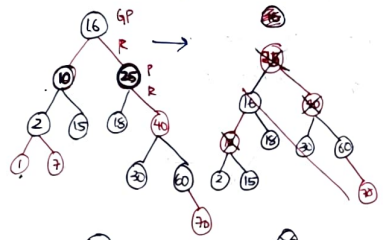


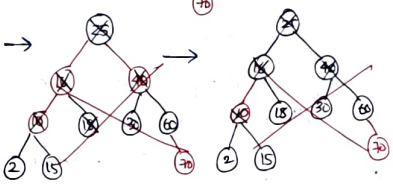

⇒ Since parent is red and sibling is null then rotation and recolouring. [parent and GP].



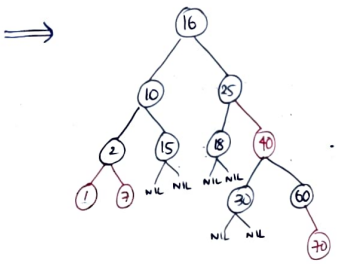

(Inserting 70)

~~X ⇒ Since the parent and sibling is red only recolouring is needed and also check GP [if not root recolour]~~



Since parent is red and sibling is red recolour it and GP is root so no change



## DELETION IN RED-BLACK TREE

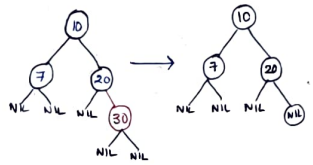
Imp: Internal nodes presence the colour

- Internal node doesnt get deleted. It is replaced with its children.

Step 1: Perform BST deletion.

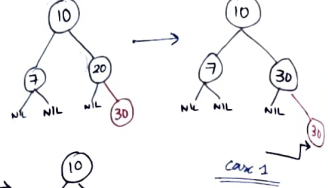
Step 2:

CASE I : If node to be deleted is red just delete it.

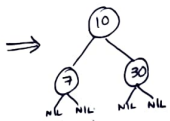


⟹ after deleting 30 we replaced it with NIL node.

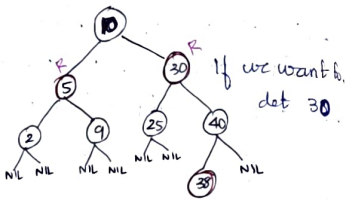
- Similarly if we want to delete 20 then it is replaced with inorder pre/succ and that child is del.



case 1



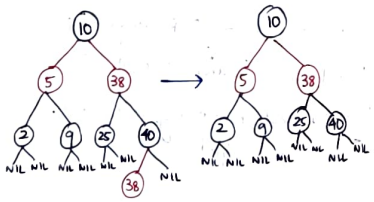
Example 2



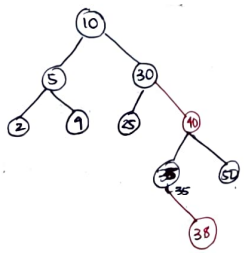
If we want to delet 30

Here it is we wanted to delete a red node with two children.

→ replace it with pred/successor. delete the pred/successor.

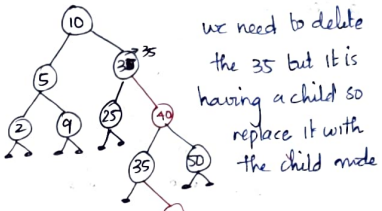


Example 3

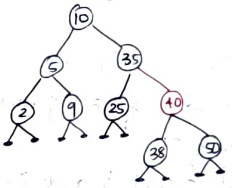


If we want to delete 30. we must replace it with pred/succ.

⟹ one point should be taken in mind that while deciding predecessor or successor we must check the no of black nodes [imbalancing]



we need to delete the 35 but it is having a child so replace it with the child node



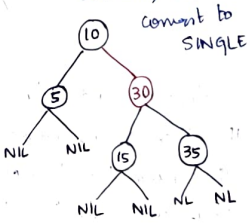


If node to be deleted is red just delete it

⟹ while we delete black we must check the path (no of black phy).

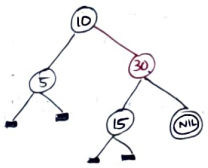
If node to be deleted is black

CASE II : ~~If root is DB just remove DB~~ replace with its black children, Double Black occurs convert to SINGLE BLACK



Case 2.1 : If Root is DB just Remove DB and make it Single black

If we delete the 35 then the NIL gets replaced at 35 there occurs double black situation.

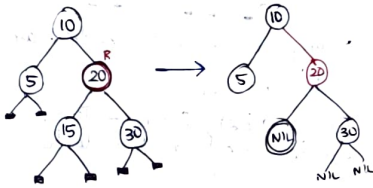


CASE 2.2 If DB's sibling is black & both its children are black

Remove DB and add extra black to its parent and make sibling as Red.

a) If parent is red it becomes black

b) If parent is black it becomes double black.

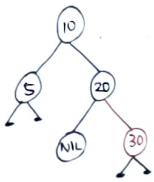


If we delete 15 and replace it with null node it becomes double black.

→ check the sibling & both its children are black

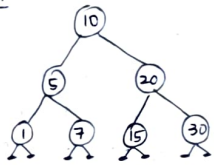
change → parent → Red [if red → black]

→ Sibling → Red.

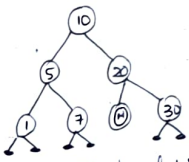



If still DB exists, apply other cases.

Ex:



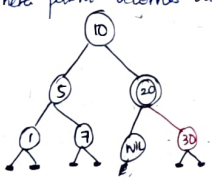

we want to delete 15. Then we must replace it with NIL children DB occurs.




Sibling and both its children are black DB/Red

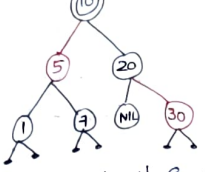
Parent → ~~Red~~ Sibling → Red :-

but here parent becomes double black

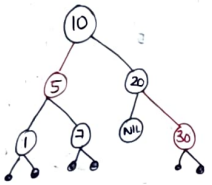



⟹ Sibling of 20 is black and both children are black. ↑ as it is black.

∴ DB is moved to parent and Sibling becomes Red.




If Root is DB make it Single black (as per rule).



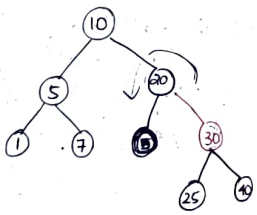

Case 2.3 :

If DB's Sibling is red.

→ Swap colours of DB's parent & Sibling of DB.

→ Rotate parent towards DB's direction.

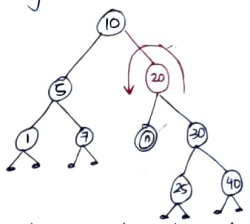
→ Reapply cases.



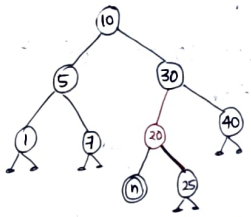

If we delete 15 and replace it with NIL then there occurs Double Black. check the sibling if black & children are black then change the DB → parent and change sib → red.

But here the Sibling is Red

So Swap the colours of DB's parent and Sibling



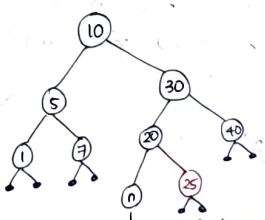

Then perform rotation towards DB




Here The Sibling of double black is black and children are black.

Then parent and sibling colours are changed

P → red → black Sibling → Red.



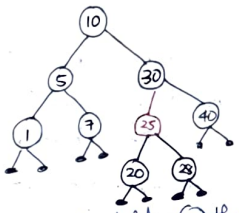

or ↓ we can just remove that.

Here what we did is Sibling of DB was black & children was black remove DB give that Black to parent & change Sibling to Red.

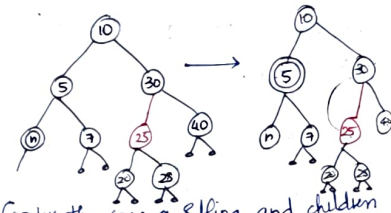
Case 2.4

DB's sibling is black, Sibling's child who is far from DB is black but near child to DB is red.

Let us take an example.




So if we want to delete (1) then we replace it with NIL node and double black case occurs.




(applys the case of Sibling and children of Sibling black).

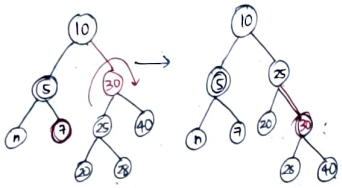
Since parent is black it becomes DB

Then we consider Sibling as black and farthest black nearest red case.

→ Swap colour with DB's Sibling and siblings child who is near to DB

→ Rotate sibling in op direction to DB node.

→ Apply case 2.5



Case 2.5

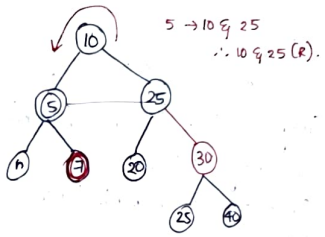
DB's Sibling is black and Sibling child who is far from DB is red but near to DB is black.

→ Swap colour of parent of DB & DB's sibling

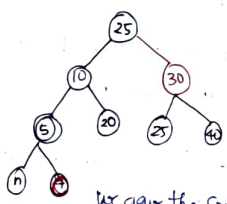
→ Rotate parent in DB's direction

→ Remove DB

→ change colour of far red child to Black.

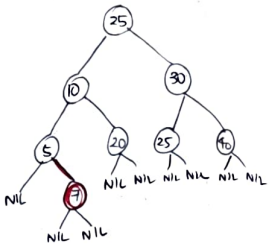



Here there will be no swapping as both are black.



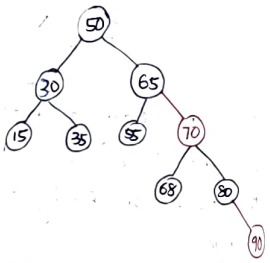
we gave the Extra black to farthest Red.

Removing the colour of far red child to black



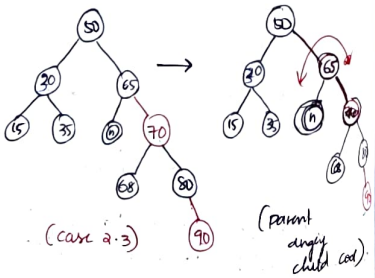
Applying All the cases.

Delete 55, 30, 90, 80, 50, 35 and 15 from.

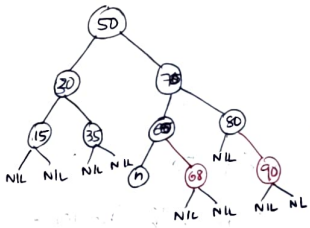


Deleting 55

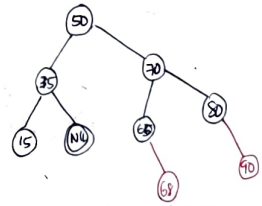
Since it is black we must think nill nodes will replace and DB occupy.



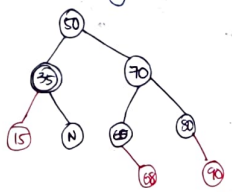

⟹ DB's Sibling is black and both its children are black. ∴ remove DB add Black to parent (if red → Black) If black → DB and Sibling → Red.



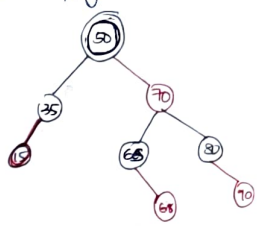
Deleting 30



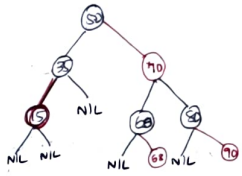
Sibling of DB is black and has black children. ⇒ parent is Black.
∴ P ⇒ DB and Sibling → Red.



→ Sibling is black and both children are black. reapply the same case.

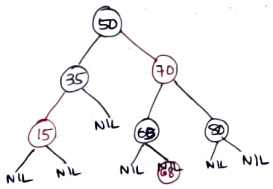


If Root is DB → Single black.



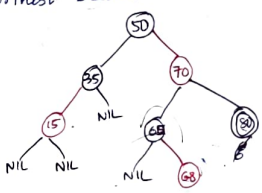
Deleting 90

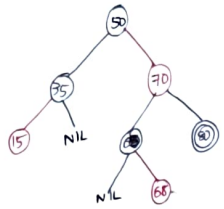
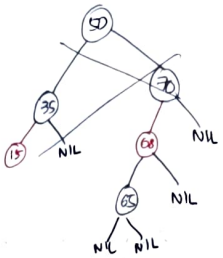
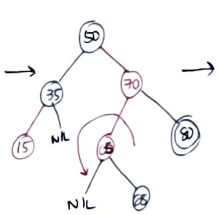
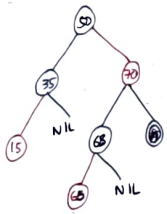
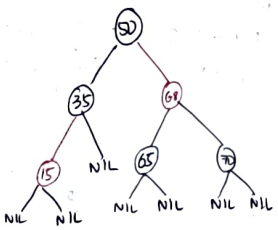
If the node is red just delete it



Deleting 80

Deleting a black node Sibling is black with nearest child → red & farthest black.



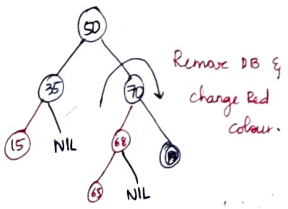
→ Sibling is black and child farthest from DB is Red.

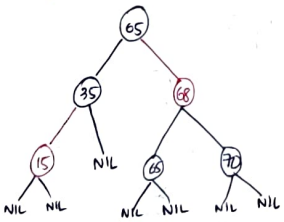
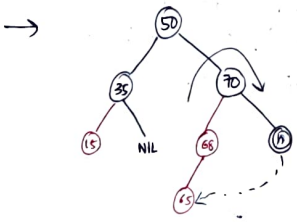
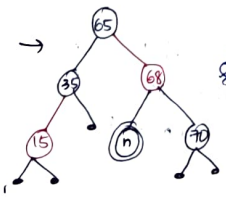
→ Parent & Sibling [swap]

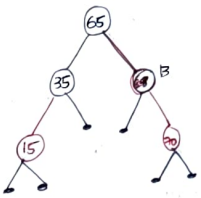
→ Rotate toward DB

Deleting 50

Replacing with Inorder successor.

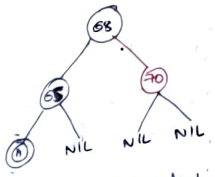


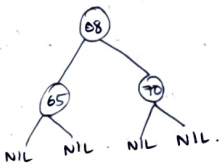



→ Since both parent and sibling is black no change.

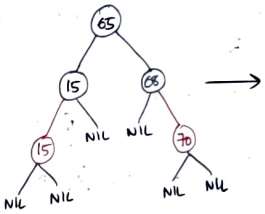
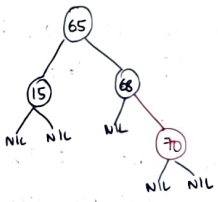
→ Rotation towards DB.



So the DB is one level down so gives the Extra black to Red one.



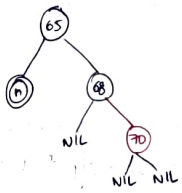
Deleting 35

Deleting 15

Check the sibling is black farthest child is red.

Parent and sibling ~~color~~ → swap.



ALL CASES.

CASE 1 : If node to be deleted is red just delete it

CASE 2 : If Root is DB Remove DB and make it single black

CASE 3 : If DB's sibling and both their children are black. Remove DB and add extra black → parent and make Sibling as Red.

parent → Red (black)
→ black (double black)

CASE 4 : If DB's Sibling is

→ Swap DB's parent & sibling colour [both same ignore]

→ Rotate parent towards DB

→ Reapply cond.

CASE 5: DB's sibling is black & sibling's child who is far from DB is red black but near child is Red

→ Swap colour with sibling & sibling's child (near DB)

→ Rotate against DB (apply case 6).

→ DB becomes one level down

→ Remove DB and change the red node.

CASE : 6

DB's sibling is black and sibling child who is far from DB is red but near is black.

→ Swap colour of parent and sibling [if same ignore]

→ Rotate parent at DB's direction

→ Remove DB

→ change colour of far red child to Black.

# SPLAY TREE

- Self adjusted binary search tree.

operations :- search, insert, delete & splaying.

- They are roughly balanced.

- time complexity of a binary search tree is

best case - O(1)

Average - O(logn) [left small, right large].

worst case - O(n) [left/right skewed]

- To limit the skewness, the AVL & RB tree came into picture having O(logn), time complexity for all operations in all cases.

- In order to improve the time complexity Splay trees were introduced.

The splay tree can be defined as the self-adjusted tree in which any operation performed on the element would rearrange the tree so that the element on which operation has been performed becomes the root node of the tree.

- Contains same operation as BST
- Has one extra ply that makes it unique ie Splaying.
- All operations in the splay tree are followed by splaying.
- not strictly balanced but roughly balanced.

There are six type of rotations.

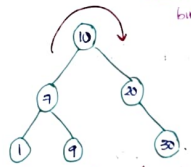
1. zig rotation (Right)
2. zag rotation (left)
3. zig-zag rotation (zig followed by zag)
4. zag zig rotation (zag followed by zig)
5. zig-zig (two right)
6. zag-zag. (two-left).

Case 1: Search item is root node.

Case 2: Search item is child of root node. (no GP).

(i) Left (ii) Right. (Search like binary search tree).

(i)

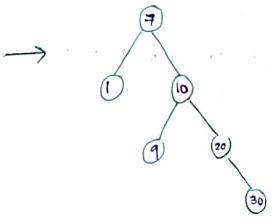



zig is performed when element is left or right

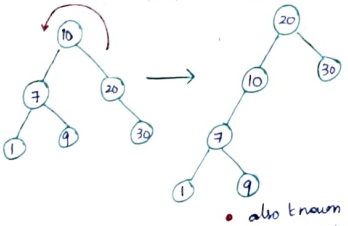
If we want to search 7. which is child of root

the left child of root node (we perform zig rotation) ie right rotation.

also known zig-right.




(ii) If we want to search 20 that is at the right of the root node so we need to perform zag rotation.




- also known as zig left rotation.

→ if node we want to search have parent and grand parent.

Case 3: If node is the right of the parent and the parent is also right of its parent (2 left rotations) are done. zig-zig (left-left)

Case 4: If node is left of the parent and the parent is left of the parent. (2 right rotation) are done. zig-zig (right-right).

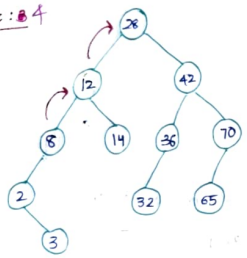
Case 5: If node is left of a parent but the node is right of the parents. (zig-zag) left right

Case 6: If node is right of parent. but the parent is at the left of its parent (zig-zag) right-left.

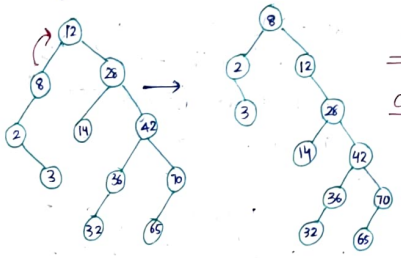
Note: zig is performed when we need to search the element which is the child of root (either left/right).

zig-zig rotation

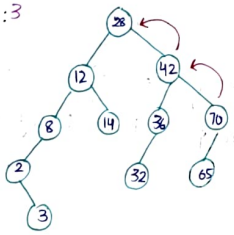
Case: 4



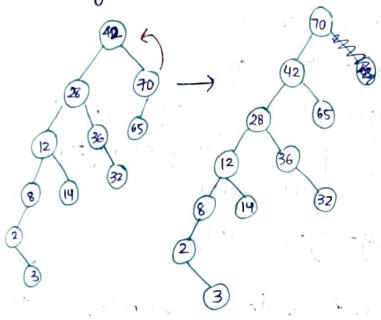
If we need to search 8. By BST searching we reached 8 but we need to perform Splaying. Here we need to perform two right rotations.



Case 3:



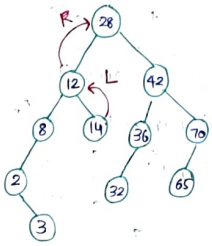
If we need to search for 70. we need to perform two left rotations inorder to make 70 to the root (i.e Splaying).



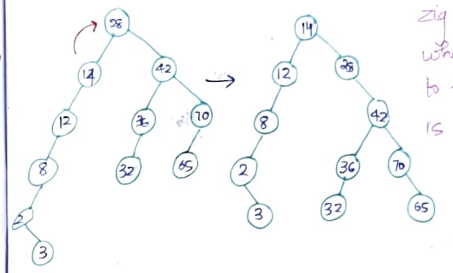
zig-zig operation is performed when direction from grandparent to parent is equal to direction from parent to the element to be searched.

zig-zag rotation.

Case 5:



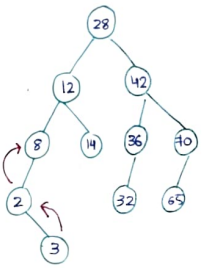
If we want to search 14 we need to perform left-Right rotation first



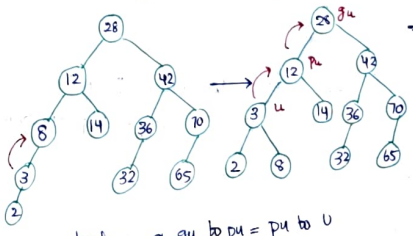
zig-zag operation is performed when direction from grandparent to parent and parent to the element is not equal [different].

```
while (u != root)
{
  if (u has no grand parent)
     do zig
  else
  {
    If (direction from gu to pu
        == pu to u)
       do zig-zig
  }
  else
  {
    do zig-zag
  }
}
```

Case 6:

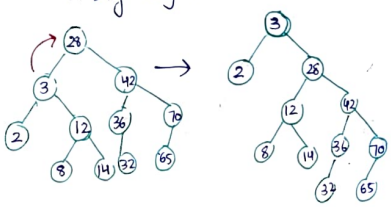


Suppose if we want to search 3 we first perform zig-zag rotation



direction of gu to pu = pu to u

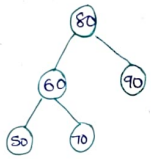
∴ zig-zig rotation.



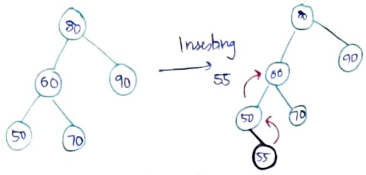
⟹ u is the element to be searched

INSERTION IN SPLAY TREES

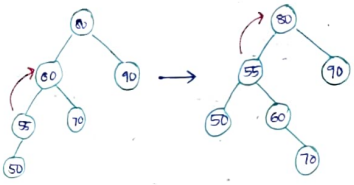
Insert 55, 45, 20, 30, 10



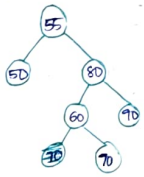
Insertion + splaying



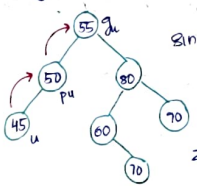

after Inserting we must perform Splaying. we perform zig-zag op.



Then we perform zig-right operation.

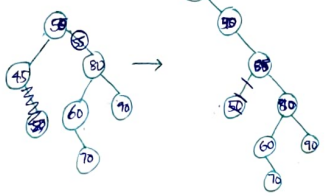


Inserting 45

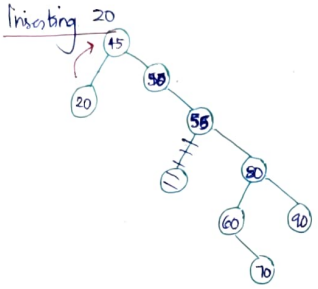


since gu to pu = pu to u

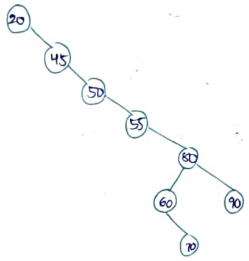
we perform zig-zig operation



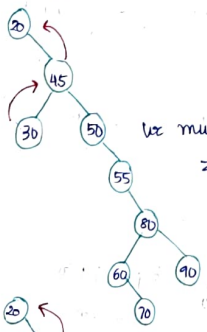
Inserting 20



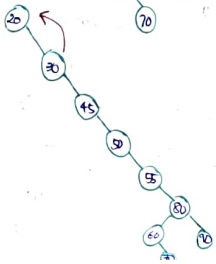
we must perform the zig operation.

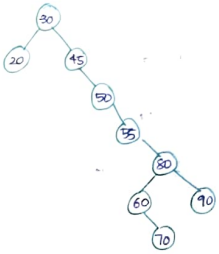


Inserting 30

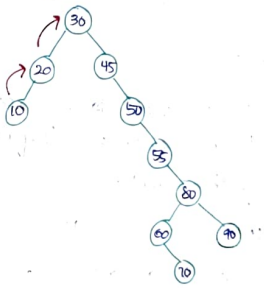


we must perform zig-zag operation.

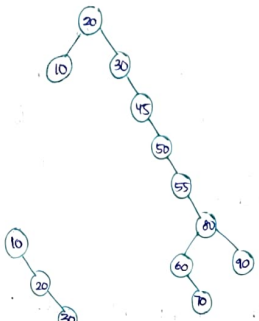


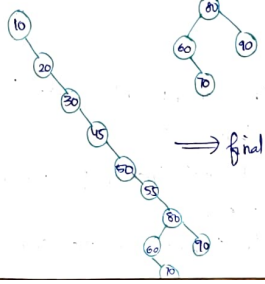


Inserting 10



Here we will perform zig-zig operation.





⟹ final splay tree.

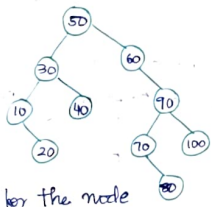
Time Complexity

Asymptotic → O(n) [worst case]

Amortised → O(log n)

DELETION IN SPLAY TREES.

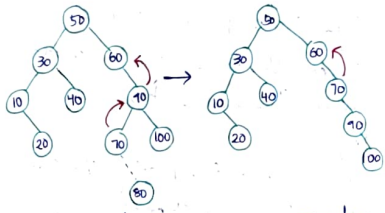
Delete 80, 30, 45 BOTTOM UP



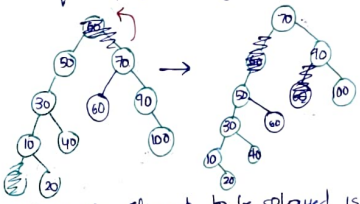
Steps

- Search for the node
- Delete that node.
- Splay the parent of deleted node.

Deleting 80

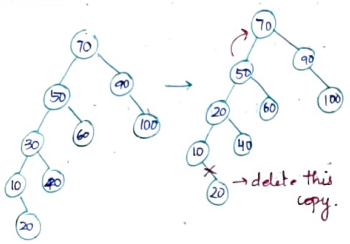


we first perform zig-zag operation.

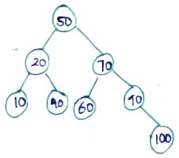


Since the element to be splayed is the child of root node we perform zig operation.

## Delete 30



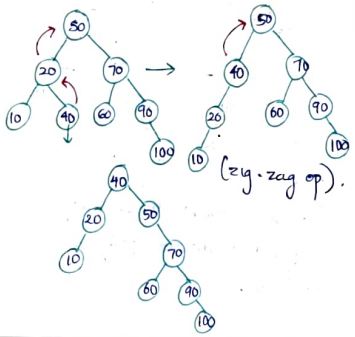

parent node = 50.
we should perform zig operation




## Deleting 45

The search of element was uncessfull. So splay the node where the search stops

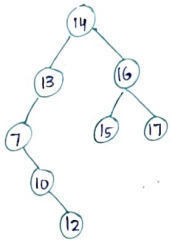
Imp
no element found splay the element where the search stopped




(zig-zag op).

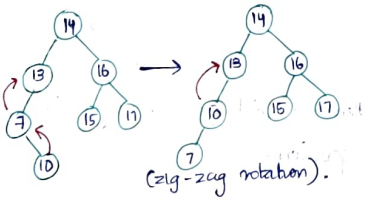
Ex 2

Delete 12, 14, 16

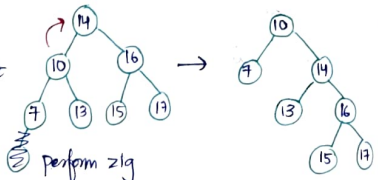



## Delete 12

case 1 : The element to be deleted is having 0 child and leaf node. So splay the parent. (10)




(zig-zag rotation).

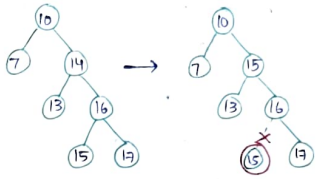



Perform zig rotation

## Delete 14

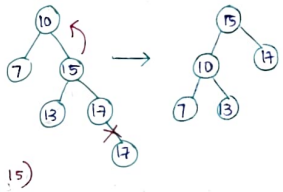
case 2: The element to be deleted is having two children

→ replace it with Inorder predecessor or successer.
→ Splay the parent of deleted element.




⇒ It is already the root so no splaying.

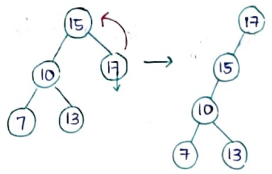
## Delete 16




(Splay 15)

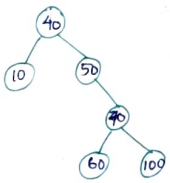
## Delete 20

case 3 : Element to be deleted is not found splay the element that ended the search.

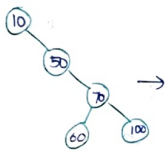



case 4 : If the element to be deleted had 2 children and the parent is already the root (no splaying).

case 5 : If we want to delete the root and we replaced it with predecessor/successor but there is no parent.



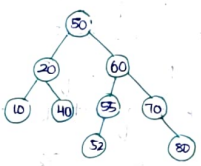

If we delete 40




→ there is no parent to splay.

## TOP-DOWN SPLAYING.

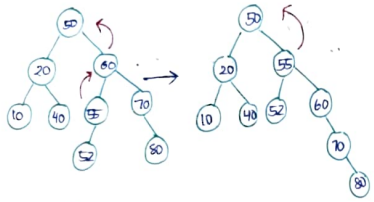
Delete 55, 50, 10, 80, 25

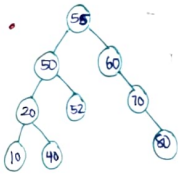



Steps

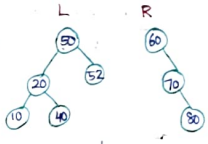
1) Search for the node
2) Splay the node so that it becomes the root
3) Delete the root
4) Perform JOIN operation.

## Delete 55





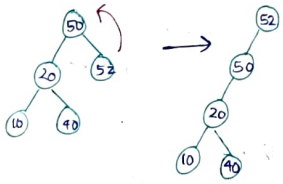
Delete the root. after that we are left with two sub trees.




- Join Operation

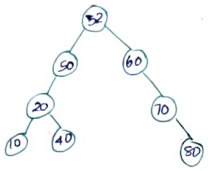
1) Splay the largest node in L so that it becomes root
2) make the root of R as the right child of root of L.

Largest node of L = 52

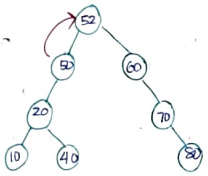


We perform zig operation to splay 52.

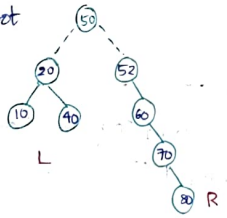
making the root of R as the right child of L:



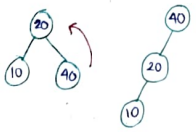
## Delete 50



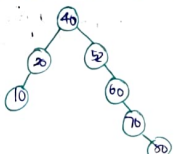
Splay the node so that it becomes the root



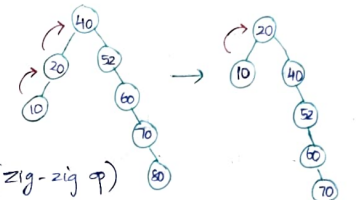

make the largest element as the root of L



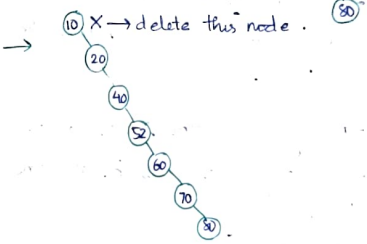
join the R as the child of Root of L



## Delete - 10



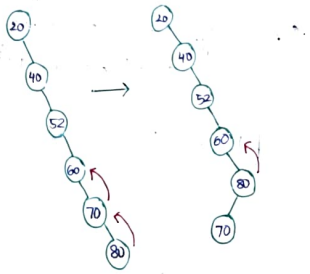
(zig-zig op)

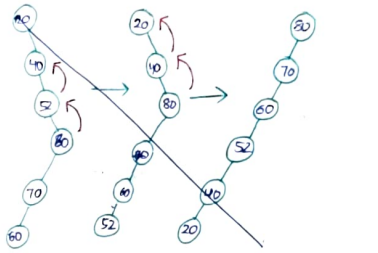


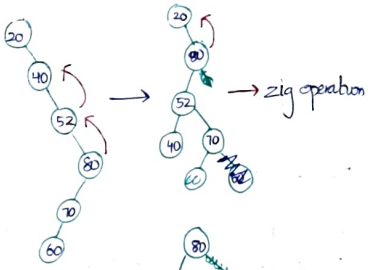

L → is null and R is only present. R subtree will be our tree as L is not there so no join operation.

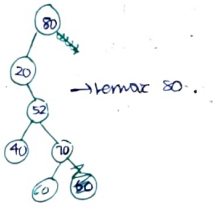
CASE 2: When L is NULL, R is the splay tree.

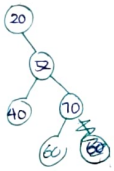
## Delete - 80







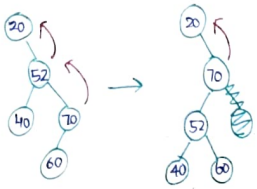



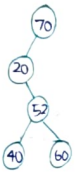



~~consider if 20 was not in this case so L would be~~

CASE 3:

When R is NULL, perform Step 1) of JOIN.
ie Splay the largest element of L

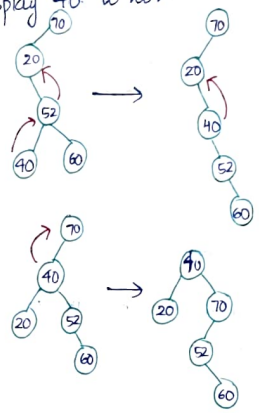




Delete - 25

we need to splay the element where the ~~element~~ search stopped.

∴ splay 40 to root.

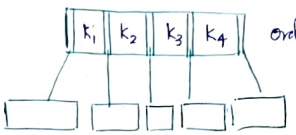




Case 4 : If the element to be deleted is not present splay the element at which the search stopped

# B - TREE

- Balanced m-way tree where m is the order.
- A B tree of order m can have at most m-1 keys and m children.
- In a B tree a node can have more than one key and more than 2 children.
- always maintains sorted data.
- all leaf node must be at same level.
- Properties. [order m]
  - Every node had max m children
  - Min children :- leaf → 0
    root → 2
    Internal node → $\lceil \frac{m}{2} \rceil$
    ie $\lceil \frac{5}{2} \rceil = \lceil 2.5 \rceil$
    $= 3$
    (ceiling).
  - Every node has (m-1) keys
  - min keys :- root node → 1
    all other node → $\lceil \frac{m}{2} \rceil - 1$

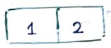



keys = m-1
nodes = m (children)

B-Tree Creation

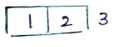
Create a btree of order 3
m=3 Insert upto values 10

max-key = (m-1) = 2




(sorted order)

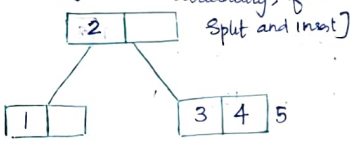
if we want to add 3



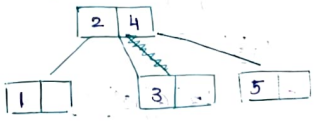

→ It is an overflow condition and we have to split the node.

→ middle element would go one level upward. [insertion → check the position, check the availability, if full split and insert]

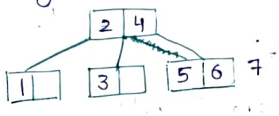
→ Inserting 4



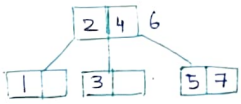

→ Inserting 5

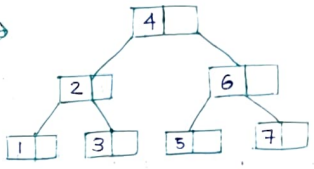



→ Inserting 6 (greater than 2 & 4)



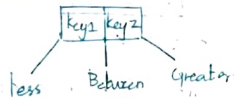

→ Inserting 7



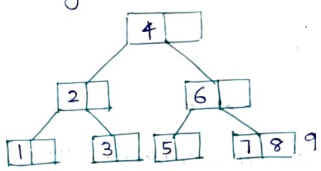



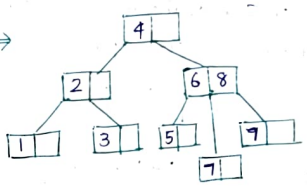

The Elements are arranged in such a manner that


Incase of two keys

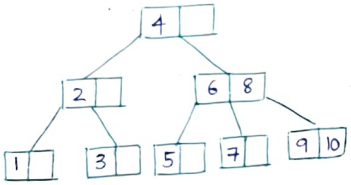



Inserting 8, 9



Inserting 10

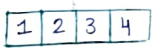



create a b tree of order 5 by inserting values from 1 to 20

$\Longrightarrow m = 5$

each node can have max = 5 children

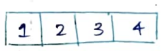
max keys = 4



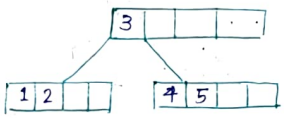

Inserting 1, 2, 3, 4

all values will be in sorted order in case of B-tree.

Inserting 5

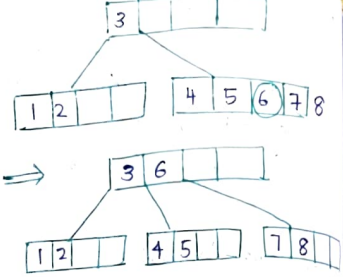



middle element / median we must consider 5 also. The middle element will move to 1 level up.

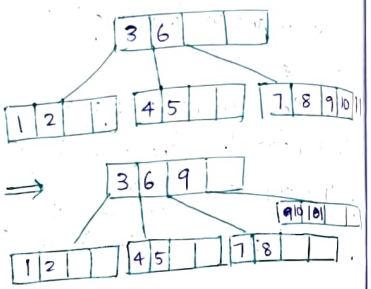



Inserting 6, 7, 8

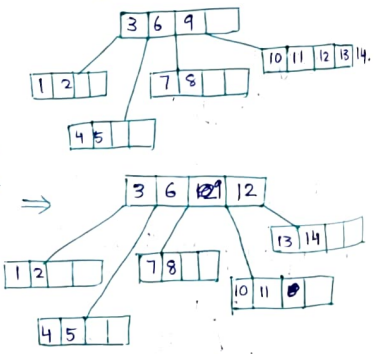
all elements are inserted in the leaf node only



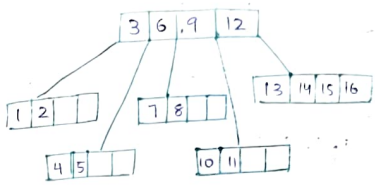

Inserting 9, 10, 11



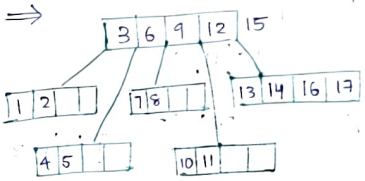

Inserting 12, 13, 14



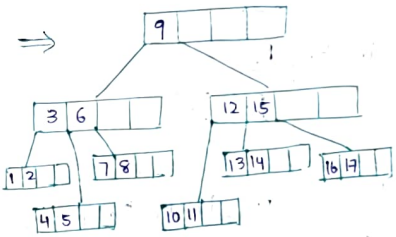

Inserting 15, 16, 17



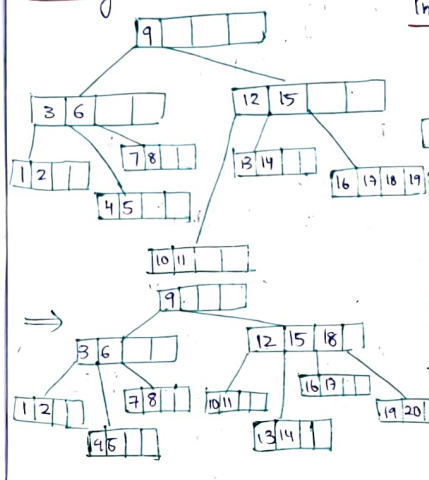

$\Longrightarrow$




$\Longrightarrow$




Inserting 18, 19, 20




$\Longrightarrow$




Q) Construct a B-tree of order 5 with the following set of data

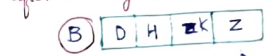
D, H, Z, K, B, P, Q, E, A, S, W, T, C L, N, Y, M

order = 5 ie $m = 5$

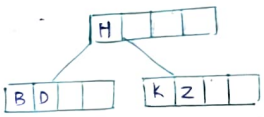
max key = 4

no of max children = 5

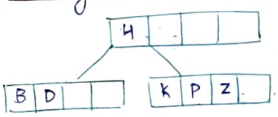
after inserting K.



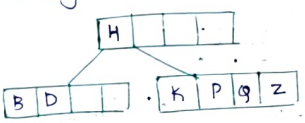

(data should be sorted)



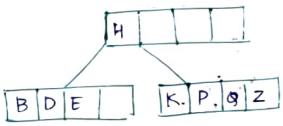

Inserting P,



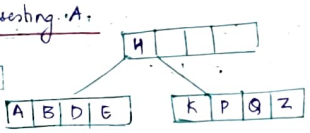

Inserting Q



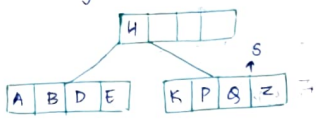

Inserting E

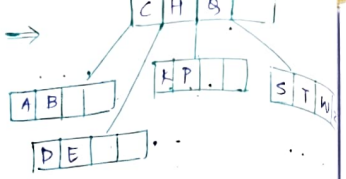


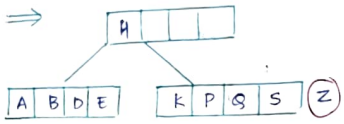

Inserting A.



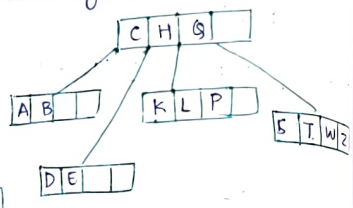
Inserting S.

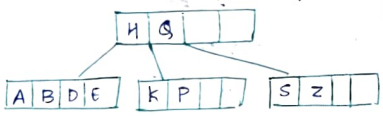


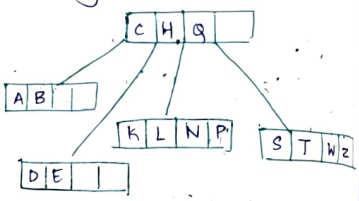




Inserting L

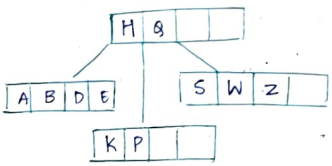


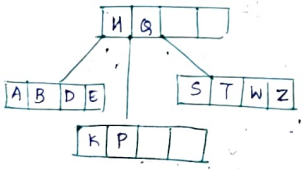

Inserting N



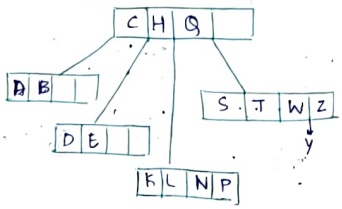

Inserting K



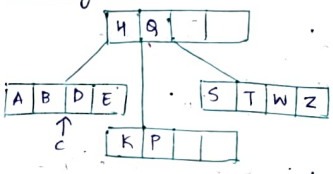

Inserting T

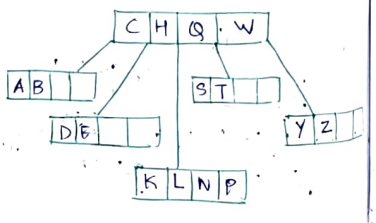



Inserting Y

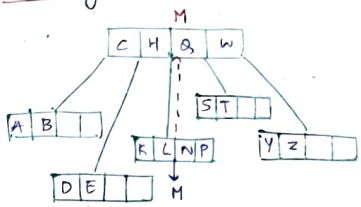


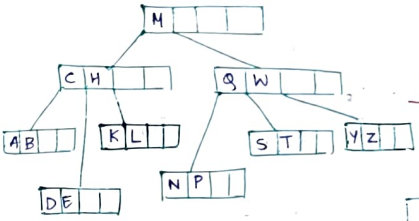

Inserting C




Inserting M



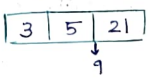

⑤ Construct a B-Tree of order 4 with the following set of data

5, 3, 21, 9, 31, 13, 2, 7, 10, 12, 4, 8.

order = 4

no of children = 4

max no of key = 3.




all the no in the B-tree stored in sorted fashion.

Inserting 31, 9

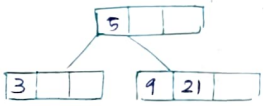
Splitting is done by finding the middle element and that middle element is moved to one level upwards.

we can take either 5 or 9 as middle element.

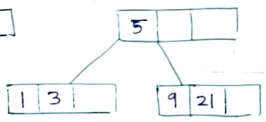
→ 5 → left biased tree

→ 9 → right biased tree.

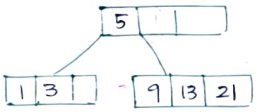
Let us take 5 as middle element



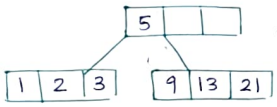

Inserting 31



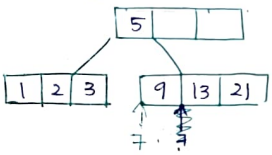

Inserting 13

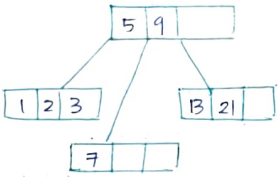


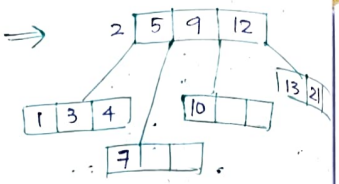

Inserting 2



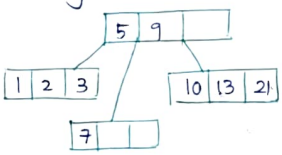

Inserting 7

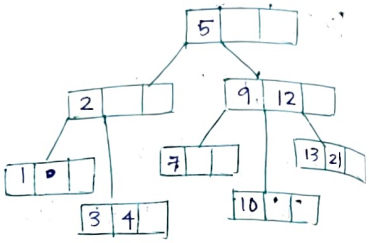




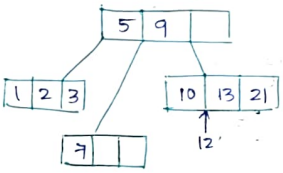



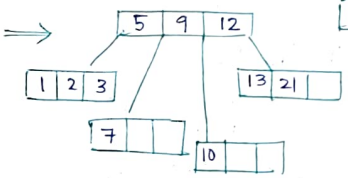

Inserting 10



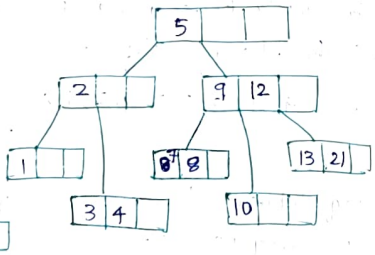




Inserting 12

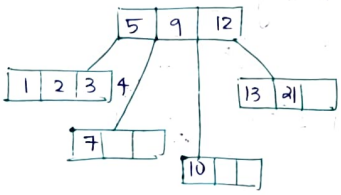



Inserting 8




Inserting 4




## DELETION OF B-TREE

CASE · 01 If target key is leaf node

① If taget key is in the leaf node.

leaf node.

that leaf node contain more than min of keys.

contains min no of keys.

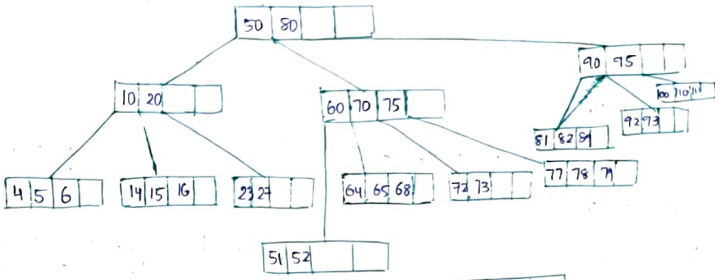
borrow from Immediate left

borrow from Immediate right

If both side don't have more than min

merge left & right with parent node

Consider the btree of order = 5




So let us take this tree and perform deletion. before that we must keep in mind that

min children = $\lceil \frac{m}{2} \rceil = \lceil \frac{5}{2} \rceil$

$= \lceil 2.5 \rceil = 3$

max children = 5

min keys = $\lceil \frac{m}{2} \rceil - 1$

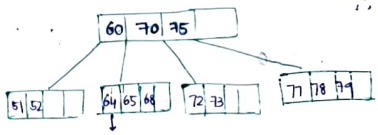
$= 3 - 1 = 2$

max keys = $m - 1$

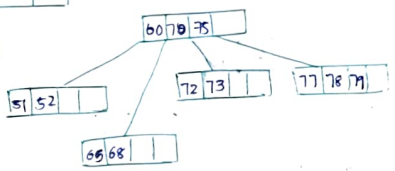
$= 4$

Eg: Deleting 64.

64 > 50 → 64 > 60 → 64



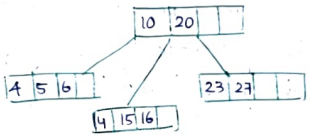

If it is on the leaf node and if the node is having more than the min keys → Then delete the element.




→ After deleting it contains min number of keys.

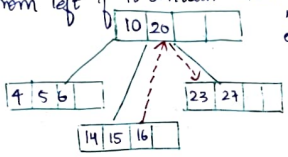
Imp: root node can contain min of 1 key

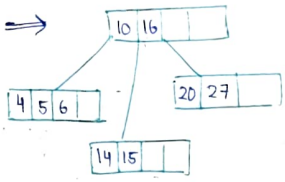
Deleting 23




→ we cannot simply delete 23 because it contains only the min number of ~~element~~ keys.

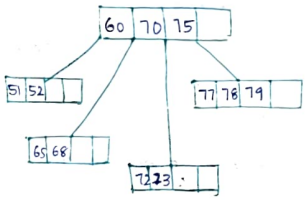
∴ we must borrow an element from left if it contain more than min no of keys.



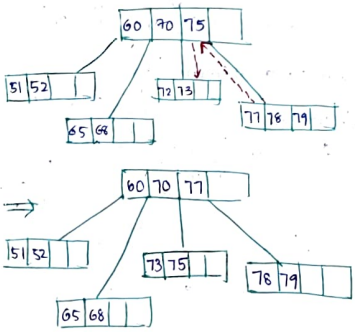



So we borrowed 20 to the parent and deleted the element.

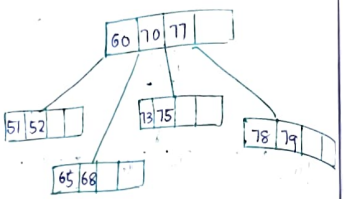
## Delete 72



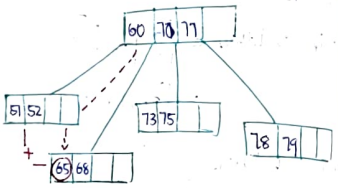

So here we want to delete 72 so it is in the minimum number of key so we must borrow an element from the right node if and only if it contains more than min no of keys.
Since it has.



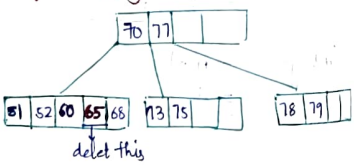

## Delete 65

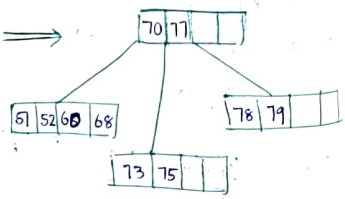



If both the neighbouring key does not have more than min key then merge either left or right node with parent also.

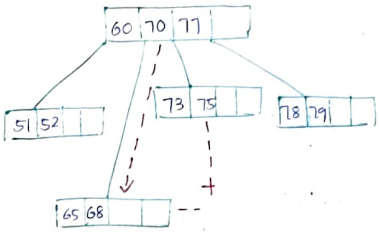


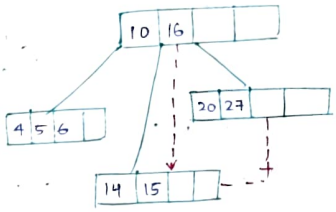

after merging

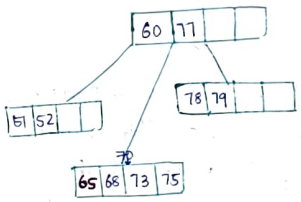


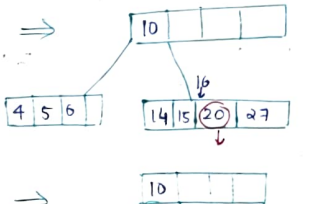



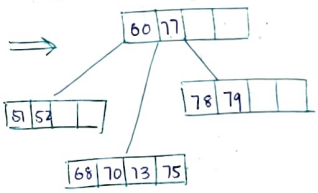

If we merge it with the right node.

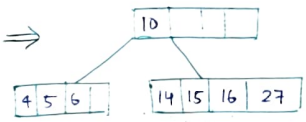


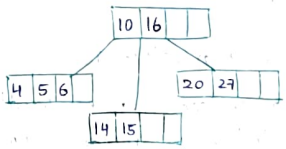



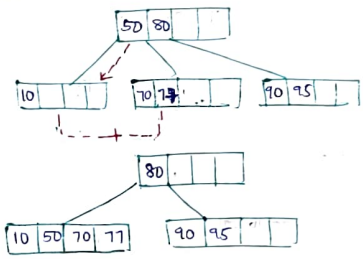




## Delete 20

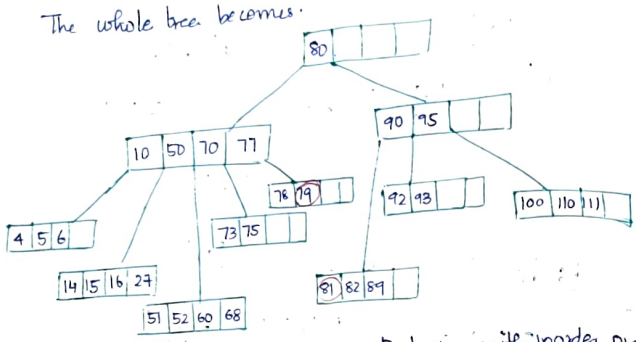



We cannot directly delete 20 because it is having min no of keys.
→ It does not have right so borrow from left but left is also having min no of keys.
→ ~~Borr~~ merge left node and Borrow Element from parent node.

But the parent node is having only 1 key which is violation of the ppty so it ~~bo~~ tries to borrow from the nearest nodes but the nearest node also have min no of keys. ie 70 77
→ now it will go to parent. parent comes down and merges the two nodes.

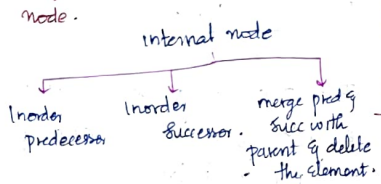


The whole tree becomes.



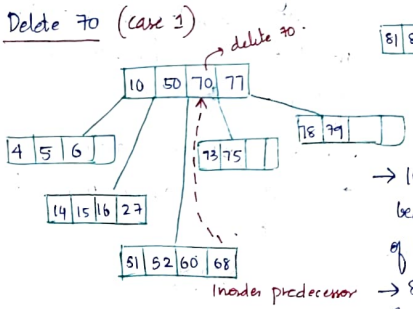

CASE · 02

If target key is in the Internal node.

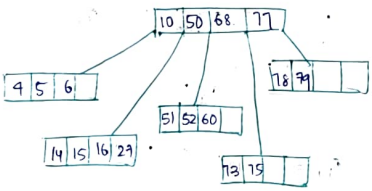



→ Replacing with inorder predecessor or successor is possible if and only if the node with the predecessor is having more than no of min keys.

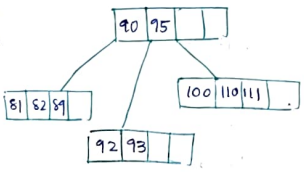
Delete 70 (case 1)




Inorder predecessor



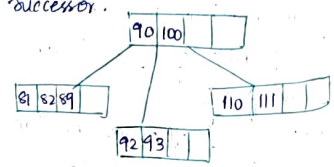

Delete 95 (case 2)




we need to replace 95

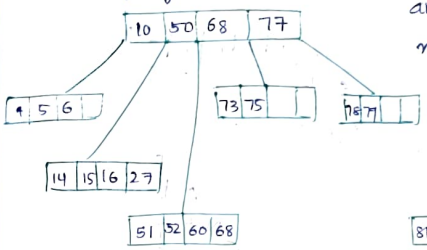
→ Inorder predecessor is not possible because it is having minimum number of keys

→ 80 we replace it with inorder successor.




Delete 77 (case 3)

⟹ If both Inorder predecessor & Inorder successor is having min no of keys.

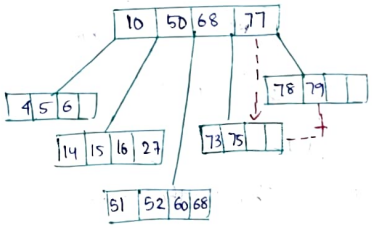


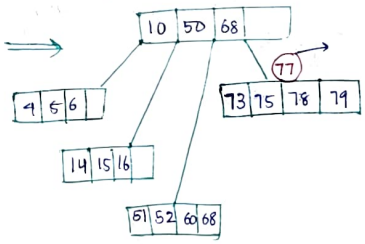

both inorder predecessor 73 75

Inorder successor 78 79

are having only min keys.

∴ merge both predecessor & successor with 77 and delete 77



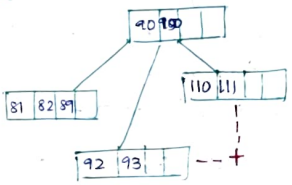



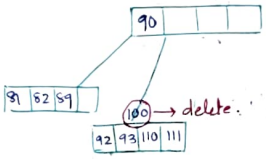

Delete 80

If we want to delete 80 replace it with Inorder predecessor.

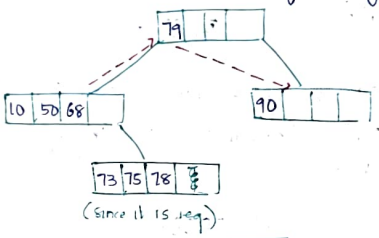
ie 79 ∴ replace it with 79 and remove 79 from the leaf node. /81

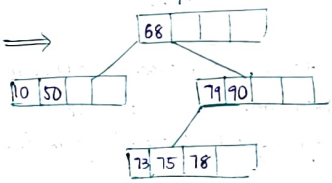
Delete 100




But 90 is in less no of min key.



Since 68 contains child node with elements when Right of 68 contains node with element 79 and 90.

The node right to 68 is sent to the left side of 79.

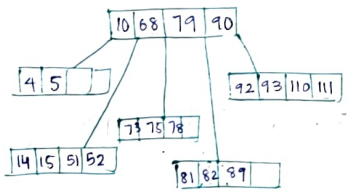
⟹ Shrinking of the B-tree.

Shrinking is caused when we delete elements from b-tree

the height is shrinked to -1.

Ie if we delete 6, 27, 60, 16 and 50 from the prev b-tree

then it will become.




The height reduced to level 1.

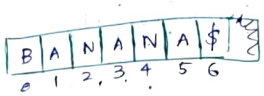
This is known as shrinking of B-Tree.

## SUFFIX TREES

- A tree data structure also known as prefix tree is a tree-like DS used for efficient retrieval of key-value pairs.

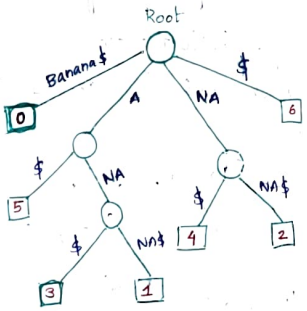
A suffix tree is a compressed tree for all the suffixes of the text

BANANA $




Suffixes are

$
A $
NA $
ANA $
NANA $
ANANA $
BANANA $




We can find whether a substring is substring or not using suffix tree

## Time Complexity

Big-O notations referred as order of is a way to express the upper bound of an algorithm's time complexity, since it analyses the worst-case situation of algorithm.

- Big-O-notation is used to describe the performance or complexity of an algorithm. Specifically it describes the worst case scenario in terms of time or space complexity.

⟹ Time complexity is not the no of milliseconds

It is the growth rate according to the increase in the no of input for ex

If an arr[] of 10 limit

If we print v = arr[0]

and print v.

Similarly we run a loop and insert value inside a array and print each values.

both have diff no of inputs

### Linear time complexity $(O(n))$

Linear complexity increase ಆಗುತ್ತಾ ಹೋಗುತ್ತೆ

ie Input ಜಾಸ್ತಿ ಆದಾಗ ಅಷ್ಟೇ time ಜಾಸ್ತಿ.

### Constant time complexity O(1)

ಎಷ್ಟೇ values ಇದ್ದರೂ ಅದಕ್ಕೆ ತಕ್ಕಂತೆ time ಜಾಸ್ತಿ ಆಗಲ್ಲ.

ie print element at arr[0]

If input is 2 or 3 or etc.. only a single value is printed.

### In nested for loop cases:

(n x n) ⟹ time complexity occurs as $O(n^2)$

for (- - - - 0), for

(0,1) (0,2) (0,3) . . . . .

n x n

f(arr[]) O(1)

for (i=0; i<n; i++) ⟹ n

Sum = sum + arr[i] O(1)

return Sum O(1)

T = O(1) + O(1) + O(1)

So n times it is calculated

∴ O(1) + n x O(1) + O(1)

const const

we consider only the other than constant values.

⟹ n x O(1)

= O(n)

Eg 2: f(arr[]) O(1)

v = a[0] O(1)

return v O(1)

O(1) + O(1) + O(1) ⟹ O(1)

O(3)

```
f(ar[][])                    O(1)
For (i=0; i<n; i++)          O(n)
    for (j=0; j<n; j++)      O(n)
        sum = sum + ar[i][j] O(1)
    return sum               O(1)
```

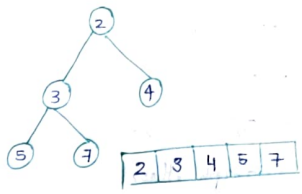
$T = O(1) + n \times O(1) \times O(n) + O(1) + O(1)$

$\Rightarrow O(n^2)$ [quadratic]

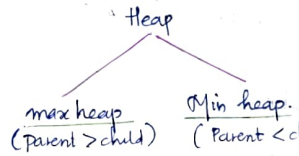
# MODULE - 03

## HEAP

Almost Complete Binary tree.



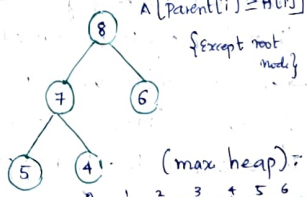

- A heap is a special tree-based data structure.
- which is a almost complete Binary tree
- it follows either maxheap or minheap.




### Max heap

The value of the root node must be the greatest among all its descendant nodes and the same thing must be done for its left and right subtrees also.

A[Parent[i] ≥ A[i]] {except root node}




### Min heap

The value of the root node must be the smallest among all its descendant nodes and the same thing must be done for its left and right subtrees also.

### Properties

- minimum and maximum element is always the root of the heap
  O(1) → time complexity.
- left child of a parent node at index 'i' is given by $2i+1$
- Right child $= 2i+2$ (indexing starts from 0)

Parent $= \left\lfloor \frac{i-1}{2} \right\rfloor$

- As it is almost complete binary tree all elements are inserted/filled except possibly the last level.
- Last level is filled from left to right.
- when we insert an item, we insert it at the last available slot of then rearrange the nodes so that the heap property is maintained.
- When we remove an item, we swap root with the last node to make sure either the max or min item is removed. Then we rearranged the remaining nodes to ensure heap property (max or min)

Applications of heap

1) Heap Sort
2) Priority Queue
3) Graph Algorithm

## Mergeable heaps

A mergeable heap performs the usual heap operations.

- make-Heap() → Returns an empty heap.
- Insert (H, x, k) → Insert an item x with key k into the heap H.
- find-min (H) → Return item with min key.
- Extract-Min (H) → Extract & Return the minimum element.
- Merge ($H_1$, $H_2$) → Combine the elements of $H_1$ & $H_2$ into a single heap

mergeable Heap (examples).

1) Binomial Heap  fibonacci Heap.

## Insertion in Max-Heap.

If we need to perform insertion we must put it in the last available pos & then check the H.P and swap.

To find the parent

$$p = \left\lfloor i - \frac{1}{2} \right\rfloor$$ (consider as array).

$A[\text{parent}(i)] \geq A[i]$ → max

min → $A[\text{parent}(i)] \leq A[i]$

## Operations

1) Heapify:

It is the process to rearrange the elements to maintain the pty of heap data structure. It is done when root is removed.

→ we replace root with the last node and call heapify to ensure that heap pty is maintained

→ or heap is build (we call heapify from the last internal node to root) To make sure that the heap pty is maintained.

2) Insertion

when a new element is inserted into the heap. It can disrupt the heap's property. To restore & maintain heap structure, A heapify operation is performed.

- This ensures the heap ptys are preserved and has a time complexity of O(log n)

3) Deletion

- If we delete the element from the heap it always deletes the root element of the tree and replaces it with the last element of the tree.
- Since we delete the root element from the heap it will distort the properties of the heap so we need to perform heapify operation that it maintains the property of the heap.

get Max (for max heap) or
get Min (for min heap)

⇒ finds the maximum or minimum element for max-heap and min-heap respectively.

Remove Min or Remove max

- This operation returns and deletes the maximum element and minimum element from the max heap and min heap respectively.

Heap tree construction.

Insert key one by one → O(n log n)

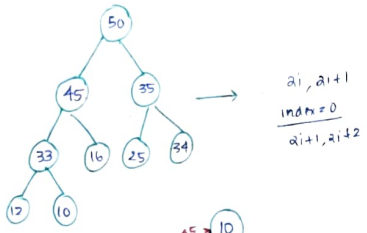
Heapify method → O(n)

## Max-Heap Insertion (just logic). not code

```
insert heap (A, n, value)
{
  n = n+1 ;
  A[n] = value;
  i = n;
  while (i > 1)
  {
    parent = ⌊i/2⌋ or ⌊i - 1/2⌋
    if (A[parent] < A[i])
    {
      swap (A[parent], A[i]);
        i = parent ;
    }
    else
    {
      return ;
    }
  } }
```

## Max-Heap deletion

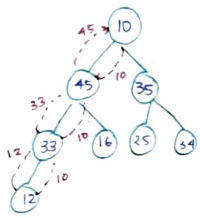
→ If we delete the root node or perform deletion root node will be deleted and the value at last position is inserted to the root.




$2i, 2i+1$

index = 0

$2i+1, 2i+2$




| 50 | 45 | 35 | 33 | 16 | 25 | 34 | 12 | 10 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

↓

| 10 | 45 | 35 | 33 | 16 | 25 | 34 | 12 | |
|---|---|---|---|---|---|---|---|---|

we have to check the max heap pty.

→ compare both the children among them the greatest would be compared with the root node.

35 and 45 → 45 will be greatest

then 45 > 10 ∴ 45 is shifted to root.

- 10 is compared with their children swapping will happen throughout. (heapify).

left child : $2 \times i = 2 \times 1 = 2$

right child $(2 \times i)+1 = 3$

$2 \times 1 + 1 = 3$

| 10 | 45 | 35 | 33 | 16 | 25 | 34 | 12 | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

$i = 1$ $2 \times 1 = 2$ 45 > 35

$(2 \times 0)+1$

| 45 | 35 | 10 | 33 | 16 | 25 | 34 | 12 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

$i = 2$ $2 \times 2 = 4$

$(2 \times 2)+1 =$

| 45 | 10 | 35 | 33 | 16 | 25 | 34 | 12 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

$i = 1$ $2 \times 1 = 2$ 33 > 16

$(2 \times 1)+1 = 3$

| 45 | 35 | 10 | 33 | 16 | 25 | 34 | 12 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

$i = 2 \times 2 = 4$

$(2 \times 2)+1 = 5$

| 45 | 35 | 16 | 33 | 10 | | | | |
|---|---|---|---|---|---|---|---|---|

| 45 | 33 | 35 | 12 | 16 | 25 | 34 | 10 | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

NB

we take from index 1.

~~If we taken from 0 it will give some errors.~~ If index = 0

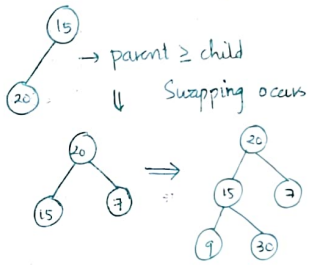
child [left] = $2i + 1$

child [right] = $2i + 2$

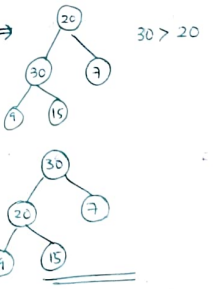
# HEAP SORT.

| A | 15 | 20 | 7 | 9 | 30 |
|---|---|---|---|---|---|

→ after each insertion we must check whether the tree is satisfying the max-heap property.

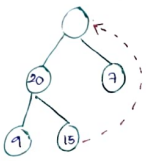



30 > 15 ⇒ 30 > 20

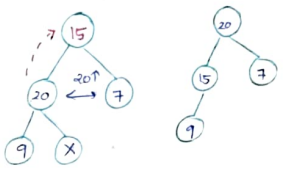



| array. | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| | 30 | 20 | 7 | 9 | 15 |

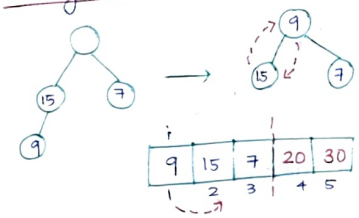
Deletion




| 20 | 15 | 7 | 9 | 30 |
|---|---|---|---|---|




Deleting 20



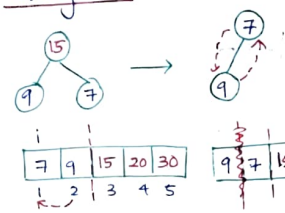

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| | 9 | 15 | 7 | 20 | 30 |

$2 \times 1 = 2$

$(2 \times 1)+1 = 3$ 15 > 7

| 15 | 9 | 7 | 20 | 30 |
|---|---|---|---|---|

Deleting 15



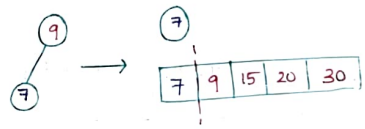

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| | 7 | 9 | 15 | 20 | 30 |

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| | 9 | 7 | 15 | 20 | 30 |

$i = 2 \times 1 = 2$

9 > 7

Deleting 9



## Deleting 7

| 7 | 9 | 15 | 20 | 30 |
|---|---|---|---|---|

⇒ This is a sorted array.

Insertion = $O(\log n) \Rightarrow O(n \log n)$ (one element ... all element)

Deletion = $O(\log n) \Rightarrow O(n \log n)$

Time complexity

$O(n \log n)$

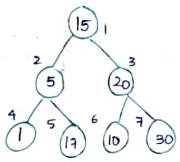
## Max Heapify

→ we need to apply heapify method on non-leaf nodes.

→ Start from right to left.

leaf node starts from

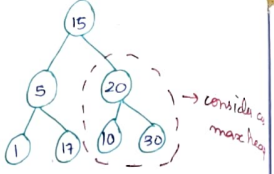
$\lfloor \frac{n}{2} \rfloor + 1$ to $n$




| 15 | 5 | 20 | 1 | 17 | 10 | 30 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

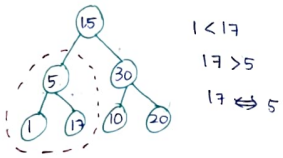
So we start Heapify method from

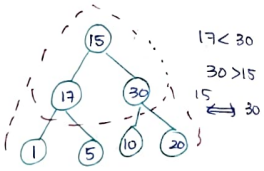
$\frac{n}{2} = \frac{7}{2} = \lfloor 3.5 \rfloor = 3$

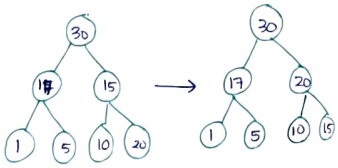



child ⇒ 10, 30 ; 30 > 20 ∴

Swapping 20 ⟺ 30




| 30 | 17 | 20 | 1 | 5 | 10 | 15 |
|---|---|---|---|---|---|---|

The time complexity would,

$O(n)$

## BINOMIAL TREES.

- It is an ordered tree.

$B_k$ - Binary tree with order k.

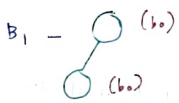
### Rules

→ $B_k$ has k children at root.

→ $B_k$ can be formed using two $B_{k-1}$ trees.

→ the root of one $B_{k-1}$ will be the leftmost child of the ~~other~~ root of other $B_{k-1}$

$B_0$ –

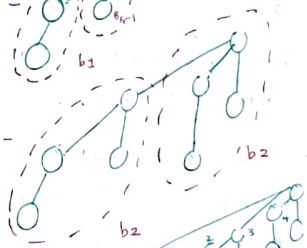


$B_1$ –




$B_2$ –


$B_3$ –




$B_4$ –

### Properties

1) There are $2^k$ nodes in a binomial tree $B_k$, height k

2) The height of the tree is k

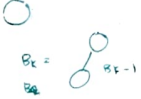
3) There are exactly $\binom{k}{i}$ nodes at depth i in a binomial tree $B_k$

4) The maximum degree of any node in a n node binomial tree is $\log(n)$

### Proof

1) $B_0 = 2^0 = 1$

$B_{k-1} = 2^{k-1}$ nodes



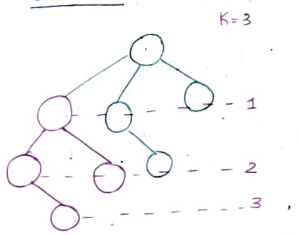

no of nodes in $B_k$

= no of nodes in $B_{k-1}$ + no of nodes in $(B_{k-1}) \rightarrow 2^{k-1}$

$2^{k-1} + 2^{k-1}$

$= 2^k$

2)




height / depth = 3

order · k = 3

$B_0$ = ○ ← 0

k=0 (height = 0)

- Let us assum

$B_{k-1}$ tree = k-1

If height $(B_2) = ?$

$B_3 = {}^{height}B_2 + 1$ (one level down for sure.)

$\Rightarrow$ height $(B_{k-1} + 1)$

∴ height of $B_k$ tree = height of $B_{k-1}$ tree + 1

∴ k-1+1 = k

3) At depth i, node of nodes in $B_k$

= no of nodes at depth i in $B_{k-1}$ + no of nodes at depth i-1 in $B_{k-1}$

$D(k-1, i) + D(k-1, i-1)$

$= \binom{k-1}{i} + \binom{k-1}{i-1}$

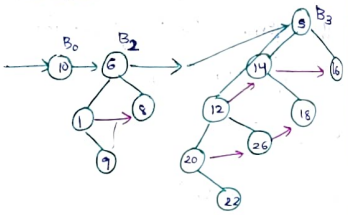
$= {}^{k-1}C_i + {}^{k-1}C_{i-1}$

$= \frac{(k-1)!}{i!(k-i-1)!} + \frac{(k-1)!}{(i-1)!(k-i)!}$

$= \frac{k!}{i!(k-i)!} = \binom{k}{i}$

## 4) BINOMIAL HEAP

- A binomial heap is a collection of binomial trees.
- A binomial tree of order 0 has 1 node. A binomial tree of order k can be constructed by taking two binomial trees of order k-1 and making one the leftmost child of the other.




★ Each and every binomial tree in a binomial heap follows min heap property.

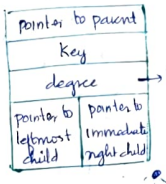
★ There should not be more than one binomial tree with same order.

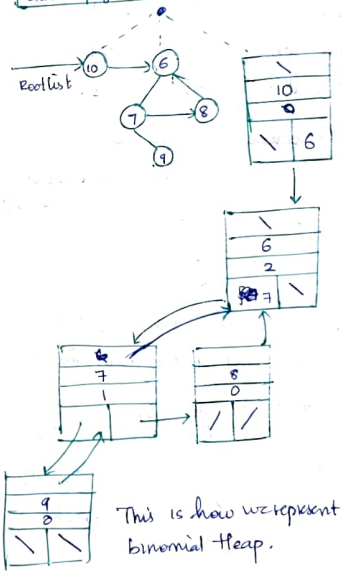
★ Binomial heap should be arranged in ascending order $(B_0, B_1, B_2, \ldots)$

(Binary Heap) ↙


Structure

### Structure of a node in Binomial Heap.




This is how we represent binomial heap.

$\Rightarrow$ If we want to combine binary trees the time complexity is $O(n \log n)$.

$\Rightarrow$ In case of binomial heap the time complexity is reduced from $O(n \log n) \rightarrow O(\log n)$.

### Operations on Binomial Heap.

→ make-Heap()

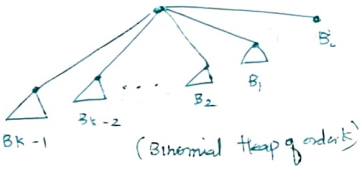
→ Insert (H, x), where x is a node to be inserted in H

→ Minimum (H)

→ Extr Exact-Min (H)

→ Union ($H_1$, $H_2$): Merge $H_1$ and $H_2$, creating a new heap.

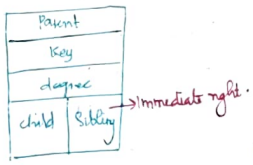
→ Decrease-Key (H, x, k): decrease x.key (x is a node in H) to k




(Binomial heap of order k)

- A binomial heap is implemented as a set of binomial trees satisfying properties
  - Each binomial tree in a heap obeys the min heap property: (The key node of a node is greater than or equal to key of its parent).
  - There can only be either one or zero binomial trees for each order including zero-order.
  - first property ensures that root of each binomial tree contains smallest key in the tree, that applies to entire heap.

- Roots of the binomial trees can be stored in a linked list, ordered by increasing order of the tree.




## MERGE OPERATION

$n = 13$

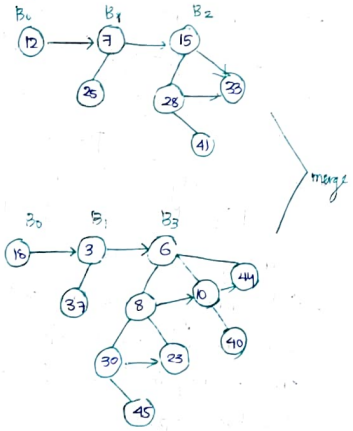
B 4 2 1 0

1 1 0 1 $\rightarrow$ $B_3 + B_2 + B_0$

$2^3 + 2^2 + 2^0$

$= 8 + 4 + 1 = 13$

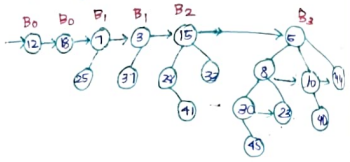
we need to merge the two binary heaps.





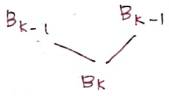

- Consider this as linked list

(1) merge both root list

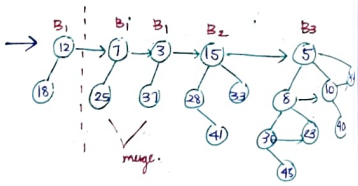


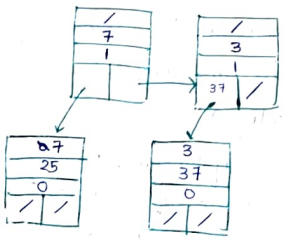

(2) we will start from the first tree and wherever we find same order we will merge those trees.

Combine binomial trees with same order (until end of root list)

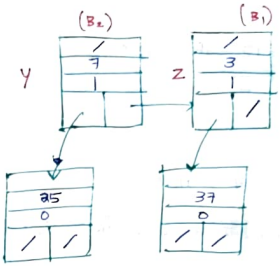



we must check which root is smaller and make them the root.



After merging we have must check which root is smaller.




follow these steps for merge op -

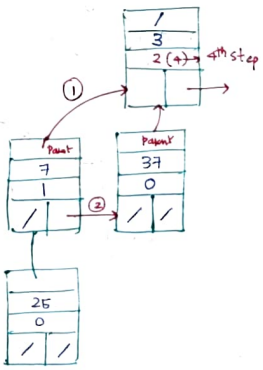
if (z > y)

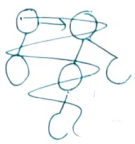
(1) parent [y] = z

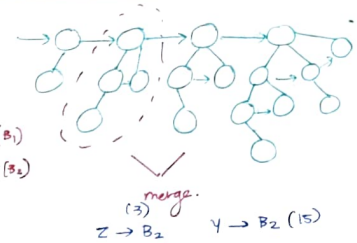
(2) Sibling [y] = child [z]

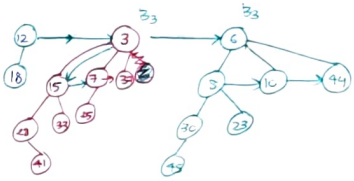
(3) child (z) = y

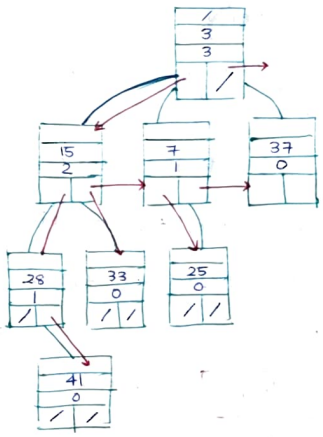
(4) degree [z] = degree [z+1]

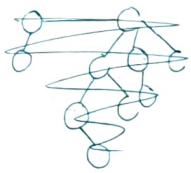


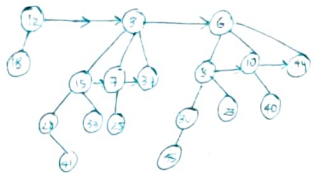



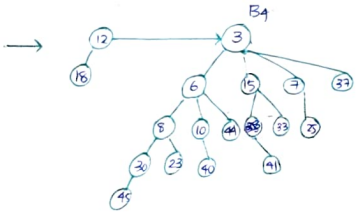







→ Time complexity

max no of nodes in binomial heap

heap = $\lfloor \log n \rfloor + 1$

n = no of nodes in binomial heap.

$TC = \max(\lfloor \log n_1 \rfloor + \lfloor \log n_2 \rfloor) + 1$

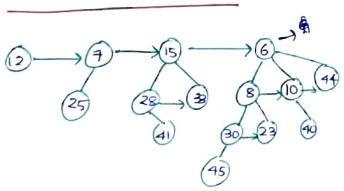
$+ 1 \times \lfloor \log n_1 \rfloor + i + \lfloor \log n_2 \rfloor + 1)$

$O(\log n) + O(\log n)$

$= O(\log n)$ where n is the

~~n~~ $n = \max(n_1, n_2)$

• find_minimum (H)



first row is known as the rootlist

min = 12, then $12 \not> 7$

$\therefore m = 7$.

$\Rightarrow 15 > 7$, min = 7

$6 < 7$, min = 6.

∴ we need to traverse the rootlist

$TC \doteq O(\log n)$

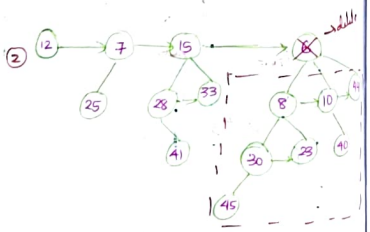
• Extract_minimum (H) $O(\log n)$

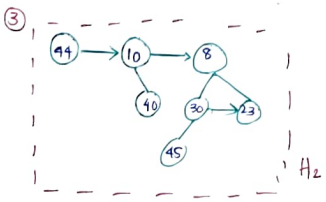
① find_min $(O \log n)$

② Delete min element node. $O(1)$

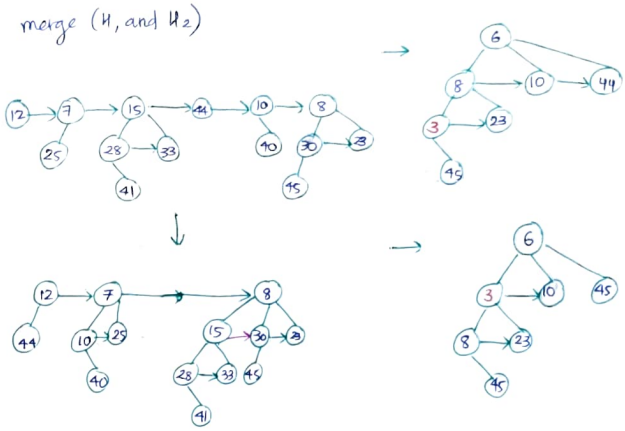
③ Reverse the root list of descendant of deleted node to get $H_2$ $O(\log n)$

④ merge $(H, H_2)$ $O(\log n)$

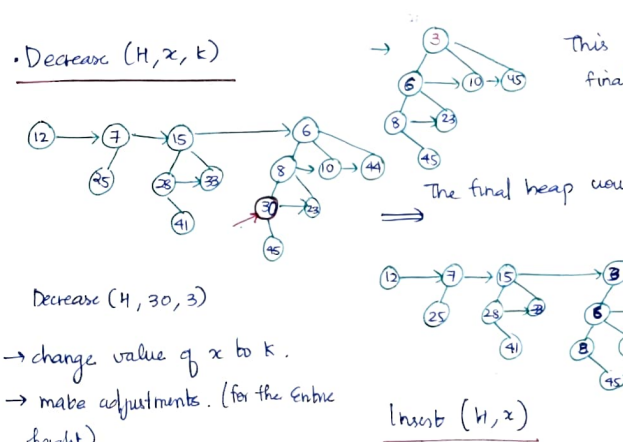





merge $(H_1$ and $H_2)$







• Decrease (H, x, k)



Decrease (H, 30, 3)

→ change value of x to k.

→ make adjustments. (for the entire height)

The entire height is the ~~root~~ $O(\log n)$

$B_k$ — height k

$\therefore$ ↳ $2^k$ nodes.

$n = 2^k$

$\log n = \log 2^k$ $O(\log n)$

$\log n = k$



This is the final adjustment

The final heap would be.



Insert (H, x)

Inserting a new element to a heap can be done by simply creating a new heap containing only this element and then merging it with the original heap. Due to this merge insert takes $O(\log n)$ time However it has amortized time of $O(1)$

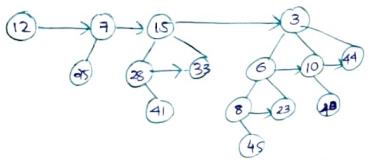
## Delete (H, x)

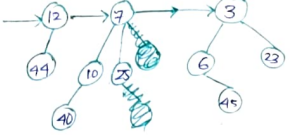
1) Decrease $(H, x, -\infty)$

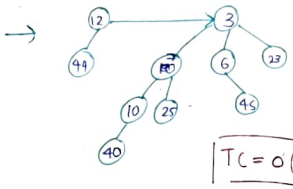
2) Delete min $(H) \rightarrow (-\infty)$ would be deleted.

3) new $H_2$ will be returned

4) merge it with $H_1$



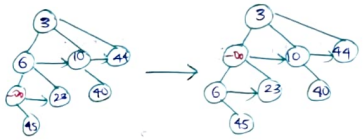


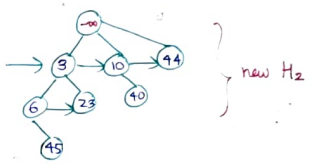


$TC = O(\log n)$

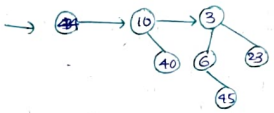


## Delete (H, 8)





new $H_2$



merge it with $H_1$

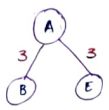
# MODULE · 04

## Graph

- A graph is a non-linear data structure that consists of vertices (nodes) and edges.
- A vertex is also called a node is a point or an object in graph
- Edges are used to connect two vertices with each other.
- Graphs are non linear because the data structure allows us to have different paths to get from one vertex to another,

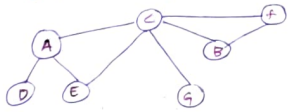
## Properties

- A weighted graph is a graph where the edges have values. The weight value of an edge can represent things like distance, capacity, time etc....

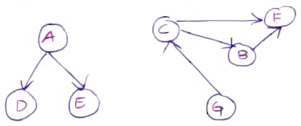


- A connected graph is when all the vertices are connected through edges somehow.

A graph that is not connected is a graph with isolated (disjoint) subgraphs, or single isolated vertices.



- A directed graph is known as a digraph, is when the edges between the vertex pairs have a direction. The direction of an edge can represent things like hierarchy or flow.



- directed cyclic

when you can follow a path along the directed edges that goes in circle.

- Undirected cyclic

when you can come back to the same vertex you started at without using the same edge more than once. (or use cyclic).

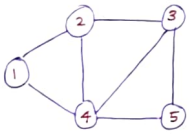
- loop is called a self loop, is an edge that begins and ends on the same vertex.

## Graph.

- Adjacency matrix
- Adjacency list

### Adjacency Matrix (better to use in dense graph)

The adjacency matrix is a 2D (array) where ~~we~~ each cell on index (i, j) stores information about the edge from vertex i to vertex j.




- Since there is no self loop the diagonal elements of the matrix will be zero (0)

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 |
| 2 | 1 | 0 | 1 | 1 | 0 |
| 3 | 0 | 1 | 0 | 1 | 1 |
| 4 | 1 | 1 | 1 | 0 | 1 |
| 5 | 0 | 0 | 1 | 1 | 0 |

Space $O(n^2)$

- It is a matrix $A[n][n]$ where n is the no of vertices

$[a[i][j] = 1$ if i and j are adjacent]

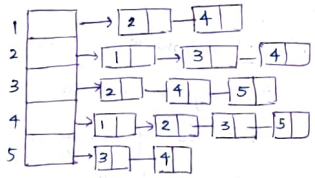
0 otherwise.

### Adjacency List (better to use in sparse graph)

In case we have a sparse graph with many vertices, we can save space by using an adjacency list compared to using an Adjacency matrix because adjacency matrix would reserve a lot of memory on empty array elements for edges that don't exist.

- Sparse graph is a graph where each vertex only has edges to a small portion of the other vertices in the graph.

- A linked list would be there for all edges (adjacent).




(Adjacency List).

Space complexity

$O(n+2e)$

## TRAVERSAL

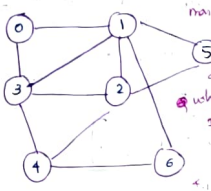
- BFS (Level order)
- DFS

### Breadth first search. (BFS)

Queue data structure is used.

- Breadth first search visits all adjacent vertices of a vertex before visiting neighbouring vertices to the adjacent vertices.
- Vertices with the same distance from the starting vertex is visited before vertices further away from the starting vertex are visited.

1. put the starting vertex into the queue
2. for each vertex taken from the queue. visit the vertex, then put all unvisited adjacent vertices into the queue. Enque the neighbour (into queue). (mark the neighbour as visited)
3. continue as long as there are vertices in the queue. Set all nodes as unvisited.



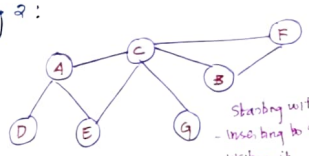

mark the starting vertex as visited and put it into empty queue
while q is not empty
1) Dequeue from v
2. Mark as visited
Enqueue w into q.
* if there is unvisited node

Queue : 0 1 3 2 5 6 4

visited : 0 1 3 2 5 6 4

when we visit a element all the adjacent ~~one~~ elements/vertices are ~~pushed~~ inserted into the queue.

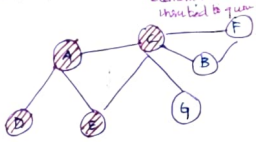
Eg 2:




BFS from D

Starting with a node
- inserting to Stack
- visiting it
- visited node adjacents are inserted to queue
- next element in queue is visited & its adj elements are inserted to queue

Queue. D A E C G B F










D, A, E, C, G, B, F.

D, A, C, E, B, F, G.

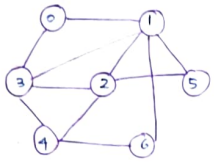
### Depth - first Traversal (DFS)

works with Stack.

- depth first search is said to go deep because it visits a vertex, then an adjacent vertex and then that ~~vert~~ vertex's adjacent vertex and so on and this way the distance from

the starting vertex increases for each recursive iteration.

1. Start DFS traversal on a vertex
2. Do a recursive DFS traversal on each of the adjacent vertices as long as they are not already visited.




| |
|---|
| 6 |
| 4 |
| 3 5 |
| 2 |
| 1 |
| 0 |

Result : ~~0,1,2,3,4,6~~

0,1,2,3,4,6,5

- any one of the adjacent vertice will be added into stack.
- after 6 there is no unvisited vertices.
- now backtracking occurs.
  (Top most element is popped)
- This process is repeated until the stack is empty.

Types of Edges in DFS. (directed graph)

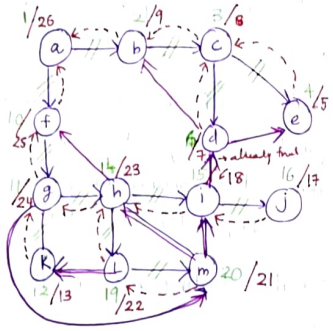
- Tree Edge : member of DFS traversal
- forward edge : $t(x,y)$ where $y$ appears after $x$ and there is to $y$.

Back Edge :

$t(x,y)$ where $y$ appears before $x$ and there is a path from $y$ to $x$.

Cross Edge :

$t(x,y)$, where there is no path from $y$ to $x$.




Tree edges : (a,b) (b,c) (c,e) (c,d) (a,f) (f,g) (g,h) (h,i) (i,j) (g,k) (k,l) (l,m) (h,l)

- forward Edge :

(x=d, y=b)

Starting time of d=6 and b=2

⇒ b appears before d

⇒ y appears before x

(b,d) ⇒ [back edge.]

- (d,e) e → d ✗

d = 6, e = 7

y appears before x.

no

⇒ There is a path from y to x

∴ ~~Back Edge~~ [Cross edge.]

- (h,f) h = x, y = f.

h = 14, f = 10 y before x.

path from y to x

f → g → h (∴ Back Edge)

- (g,m) x = g (11), y = m (20)

x comes before y.

⇒ There is a path from x to y

[forward Edge]

- (m,i) x = m (20), y = i (15)

y comes before x

⇒ There is no path from y to x

[cross Edge]

- (i,d) x = i (15) y = d (6)

y before x

⇒ There is no path from y to x

[cross edge]

- (m,h) x = m (20), y = h (14)

y appears before x

⇒ There is a path from y to x.

[back edge]

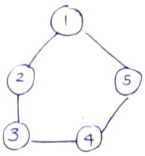
forward Edge : (g,m)

Back Edge - (d,h), (h,f) (m,h)

Cross Edge - (d,e), (m,i) (i,d) (k,l)

## MINIMUM SPANNING TREE

- The minimum spanning tree is the collection of edges required to connect all the vertices of an undirected graph, with the minimum total edge weight.




G(V,e)

G'(V',e') ↓ Spanning tree

V' = V

- In a spanning tree all the vertices will be same.

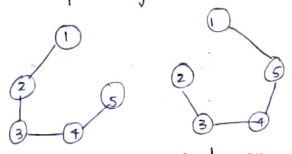
e' = subset of e

e' ⊂ e

e' = |V| - 1 (no of edges in a spanning tree would be no of ~~edges~~ vertices in a graph - 1)

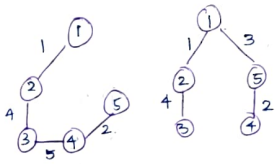
- A graph can have more than 1 spanning tree.

The spanning trees are.



and many more.

→ and if there is weights on Edges. The total of the edges of the spanning tree which is minimum is the minimum spanning tree.



1+4+5+2 (12) cost

1+4+2+3 (10) cost

(Minimum Spanning Tree)

⇒ The spanning tree should not contain any cycles.

⇒ Spanning tree should not be disconnected.

## Properties

→ Removing one edge from the spanning tree will make it disconnected.

→ Adding one edge to the ST will ~~not~~ create a loop

→ If each edge has distinct weight then there will be one & unique MST

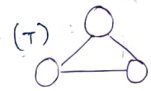
→ A complete undirected graph can have $(n^{n-2})$ (cayley's theorem) no of ST

→ Every connected & undirected graph has atleast one ST

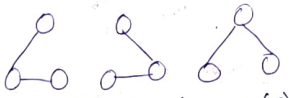
→ Disconnected graph does not have any ST

→ from a complete graph by removing max $(e-n+1)$ edge we can construct a ST

(complete graph: all vertices are ~~connected with each other~~ having an edge)



(T) Complete graph



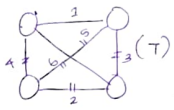
Spanning trees of (T)

$e = 3,\ n = 3$

$\therefore 3 - 3 + 1 = 1$

By removing 1 element we can create spanning tree.

Eg: 2

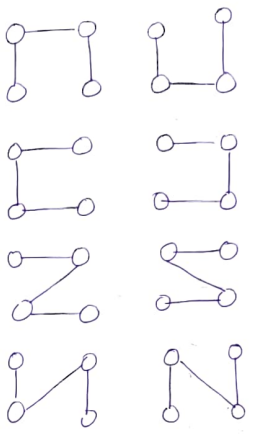


(T)

max ST = $n^{n-2}$

$4^{4-2}$

= 16 trees



etc.....

no of Edges to be removed.

edges = 6 − 4 + 1

= 2 + 1 = 3

We can remove maximum of 3 Edges.

## PRIM'S ALGORITHM

1. Remove all the loops and parallel Edges.
2. choose a random vertex as the starting point, and include it as the first vertex in the MST.
3. compare the edges going out from the MST, choose the edge with the lowest weight that connects a vertex among the MST vertices to a vertex outside the MST
4. Add that edge and vertex to the MST
5. keep doing Steps 3 and 4 until all vertices belong to the MST.

ALGORITHM

Step 1 : Select a starting vertex.

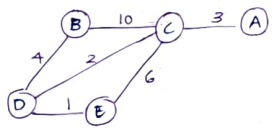
Step 2 : Repeat Steps 3 and 4 until there are fringe vertices.

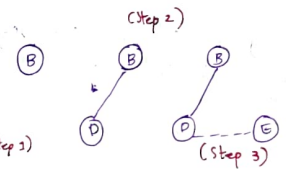
Step 3 : Select an edge connecting the tree vertex and fringe vertex that has minimum weight

Step 4 : Add the selected edge & the vertex of the minimum spanning tree T

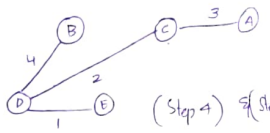
[End of loop]

Step 5 : Exit.





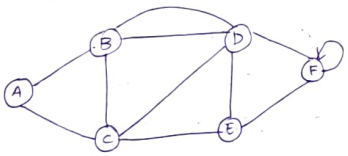
(Step 3) (Step 2) (Step 3)



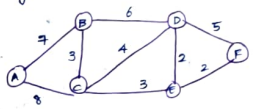

(Step 4) & (Step 5)

Example 2

Construct a minimum spanning tree from the graph given below.

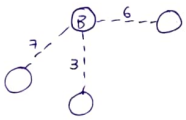



- removing the loops and parallel edges.



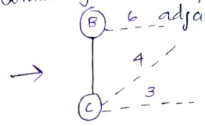

- considering B as the starting vertex

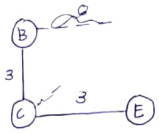

- considering all the adjacent vertices:

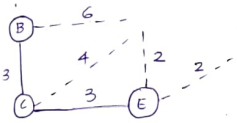


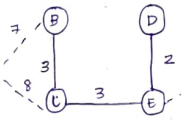

- comparing the cost and choosing the edge with low cost

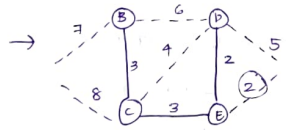
considering other 6 adjacent edges.

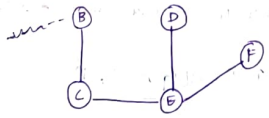


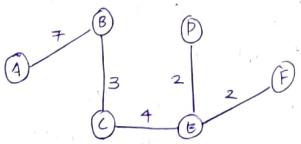




(MST using prims Algo)

## KRUSKAL'S ALGORITHM

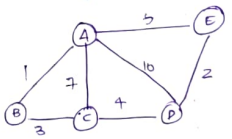
Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which, we can traverse every vertex of the graph.

- follows greedy approach which finds an optimum solution at every stage instead of focusing on a global optimum.

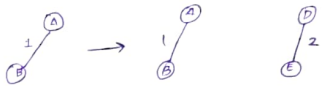
Step 1: Remove all loops and parallel edges.

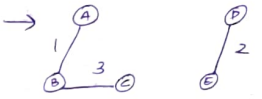
Step 2: Arrange all the edges in the increasing order of their weight

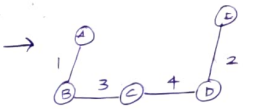
Step 3: Add the edges which have least weight.

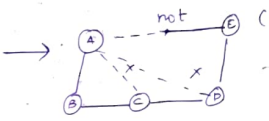



AB = 1
DE = 2
BC = 3
CD = 4
AE = 5
AC = 7
AD = 10

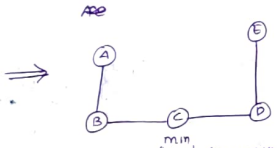



Ade

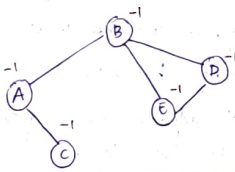



min
(cost spanning tree).

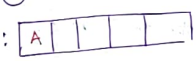
we cannot add AE & AC and AD since it forms a cycle.

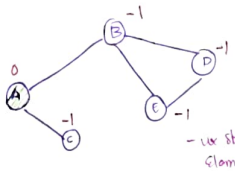
## Detect cycle in Undirected Graph

→ using BFS traversal.




flag = -1
if flg = 0 then it is put already in queue
1 = traversed.

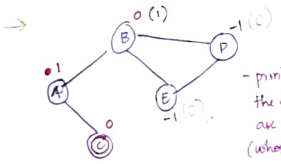
Queue : 



- we start with an element insert it into queue print it and insert the adjacent nodes into queue

Queue: A | B | C | | |

Visited: A,

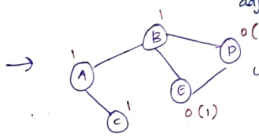


- printed B and the adjacent nodes are D and E (whose flag = -1) insert it into Queue.

Queue: A | B | C | D | E

Visited: A, B,

⇒ If any vertex finds adjacent vertex with flag 0 then it contains cycle.



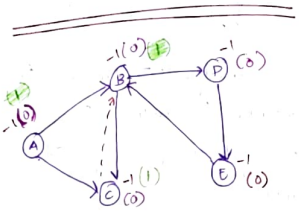
Queue: A | B | C | D | E

Visited: A, B, C, D, E

- Here it is started with an element pushed into the queue. and traversing that element.
- Inserting the adjacent nodes of the visited node whose flag = -1 is inserted into the queue.

- elements inserted into the queue flags are changed to 0.
- elements visited flag = 1
- After traversing all the elements ie flag = 1 for all elements the traversing is stopped.

## Detect cycle in ~~Undirected~~ Directed Graph



BFS structure we use stack:

Visited set

A, B, C, D, E

Flag

-1 = unvisited
0 - visited & in stack
1 = visited & popped from stack

Parent Map

| Vertex | Parent |
|---|---|
| A | - |
| B | A |
| C | B |
| D | B |
| E | D |

→ Here after C there is no adjacent nodes it is a dead end so we need to backtrack (pop) from stack

→ The next element is B

→ If we reach a dead end remove the top of the stack.

→ A & E we have an adjacent node but it's flag is already (0)

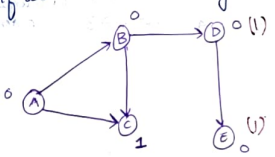
If any vertex finds an adjacent vertex with flag 0 then the graph is having a cycle.

h (E→B)

Parent of E ⟶ D

B → D → E → B → cycle.

If we remove the edge B,E.

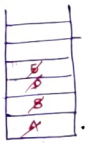


→ So at E there is no Adjacent.

→ Pop E

→ ~~At D Adjacent is B~~ (~~But it is already~~ 0)

~~Parent of (B → D)~~



~~A → B~~

→ In D there is no Adjacent vertex with flag = 1 (pop D)

→ B (no Adjacent vertex with flg = 1) pop (B).

→ A → no vertex with flag = 0 ∴ -1 ∴ remove A.

Stack is empty. O(V+E).

## TOPOLOGICAL SORTING

→ It is a linear ordering of its vertices such that for every directed edge uv for vertex u to v, u comes before vertex v in the ordering.

Graph should be DAG

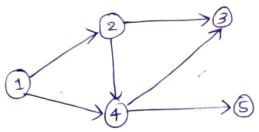
Every DAG will have atleast one topological order.

DAG → Directed & Acyclic graph.

Imp u ⟶ v

u must come before v.

Eg:



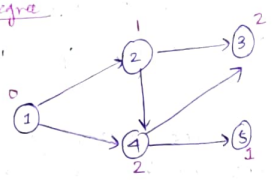
Acyclic and Directed graph.

Step 1: Find Indegree and outdegree of each vertex

Indegree: no of edge coming to that vertex.

Outdegree: no of edges going from that vertex.

Indegree

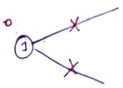


We will choose the vertex with Indegree 0.

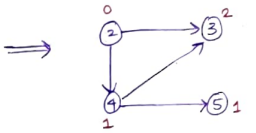
1 → 2 → 4 → 3 → 5

Step 2

We will delete all the outgoing edges from that ~~e~~ 1

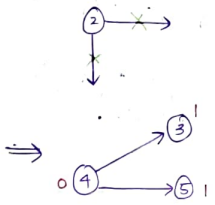


If we delete that edges the indegree of other ~~elements~~ vertex would be changed.



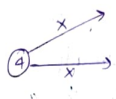
Select the vertex having Indegree 0. ⇒ 2

delete the outdegree vertex.



Select the vertex having Indegree

(4)



⇒

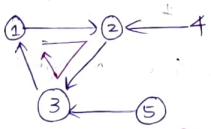
Vertex having Indegree 0

3 and 5. (Delete 3 and 5).

1 → 2 → 4 → 3 → 5 } Topological order

1 → 2 → 4 → 5 → 3 }

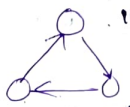
Eg: 2



(cyclic) not possible.

→ If there is a 0 Indegree we consider it as 2 cases.

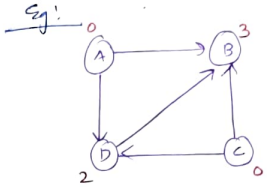
→ In case of cyclic graph At some point it will become like



Indegree would be 1 for every vertex

⇒ further procedure is not possible.

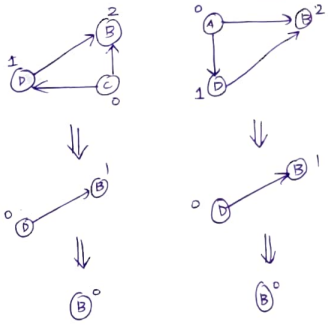
Eg:



A → C → D → B

C → A → D → B

Case 1



Case 2



Topological sorting

① A → C → D → B

② C → A → D → B