PAGE NO.:

Unit-4

Advanced tree.

* Threaded binary tree
  - Creation & traversal of in-order, pre-order & post-order threaded binary tree
  - Insertion & deletion of nodes in threaded binary tree

* AVL trees

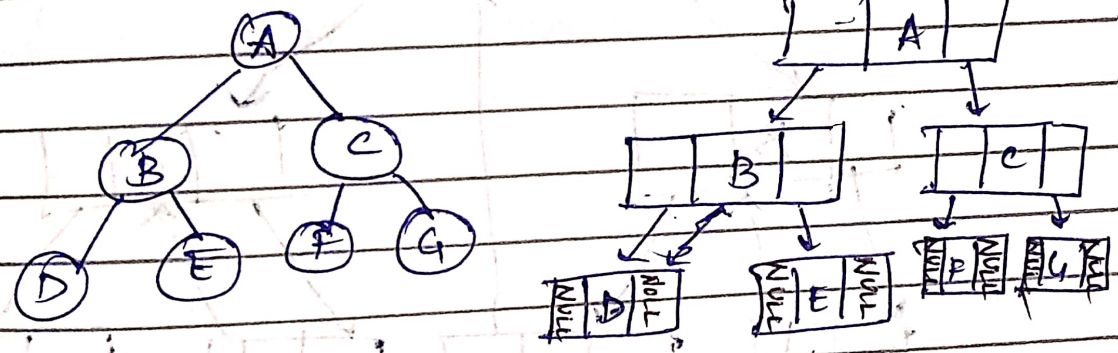
* Creation of Heap tree and Heap sorting

* Huffman tree.

Threaded binary tree (TBT)

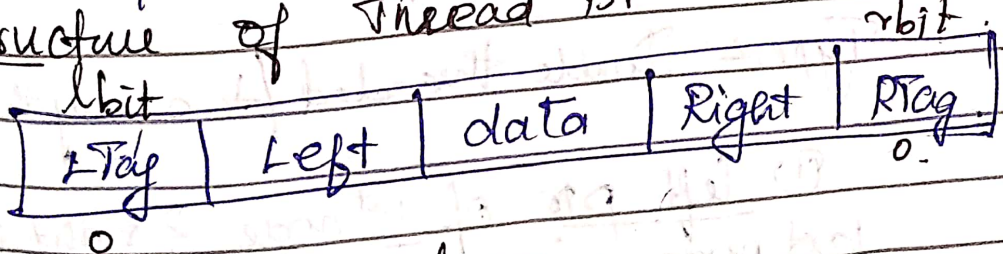
- A binary search tree in which each node uses an otherwise-empty left child link to refer to the node's inorder predecessor & an empty right child link to refer to its in-order successor

Simple binary tree : Threaded binary tree



TBT mainly used for avoid ptrs where info can be stored.
Uses a special pointer → called thread. Used to pt to higher nodes.

Node Structure of Thread BT



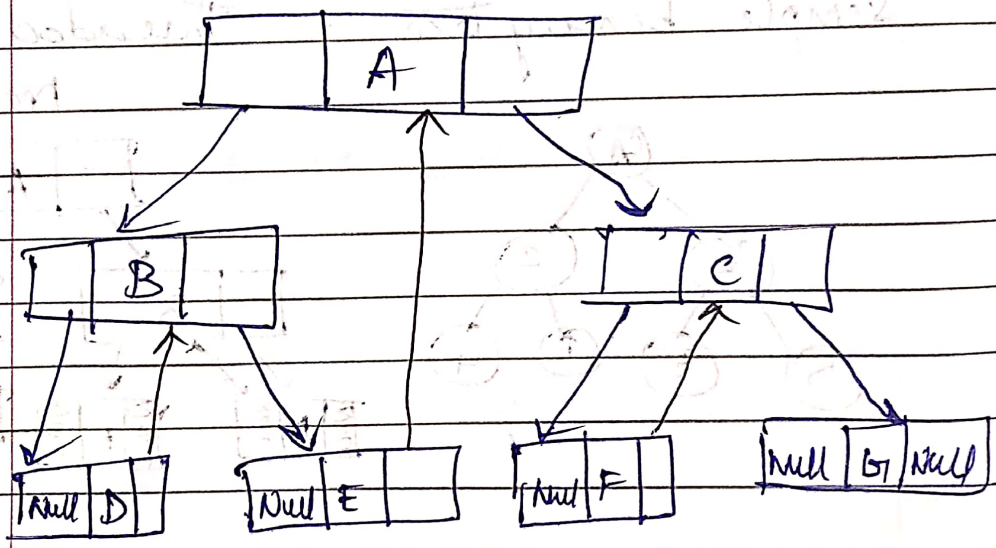
Types of threaded BT

1. Single threaded / One way threading.
2. double threaded / Two way threading.

Type 1: Single threaded / one way threading.

(1) Right threaded are ~~used~~ only used here

(2) Implementation: ~~used~~ in-order successor of tree is used here.



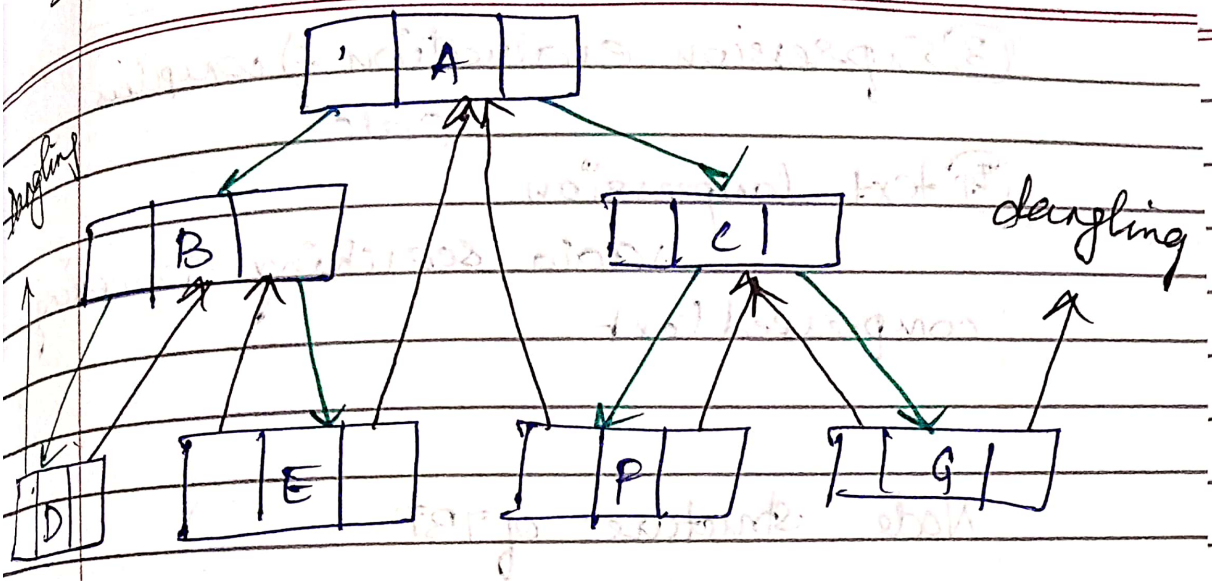
Inorder traversal of the tree:

D B E A F C G.

Type 2: Double threaded / two way threading

(1) Left ptr of 1st node & right ptr of last node will contain NULL value

Inorder traversal: D B E A F C G.



* predecessor threads are useful for reverse inorder & postorder traversal.

Uses of threaded binary tree:
↳ memory efficiency + traversal speed

⇒ to traverse a binary tree a binary tree w/o using recursion / stack which can help in improve traversal efficiency & save memory.

⇒ require space of threaded ptr, but it will have improve traversal performance.

Real world examples:
① Text editors
↳ structure of docs.
features: searching, highlighting.

② File s/y. ⇒ store & manage directory structures

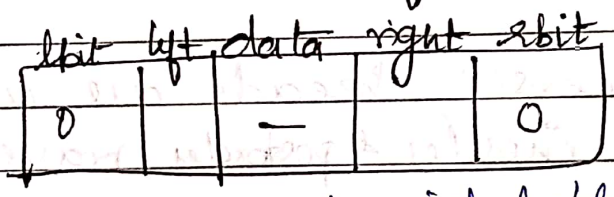
PAGE NO.:

(3) Expression evaluation :) compilers, calc

(4) Text compression.

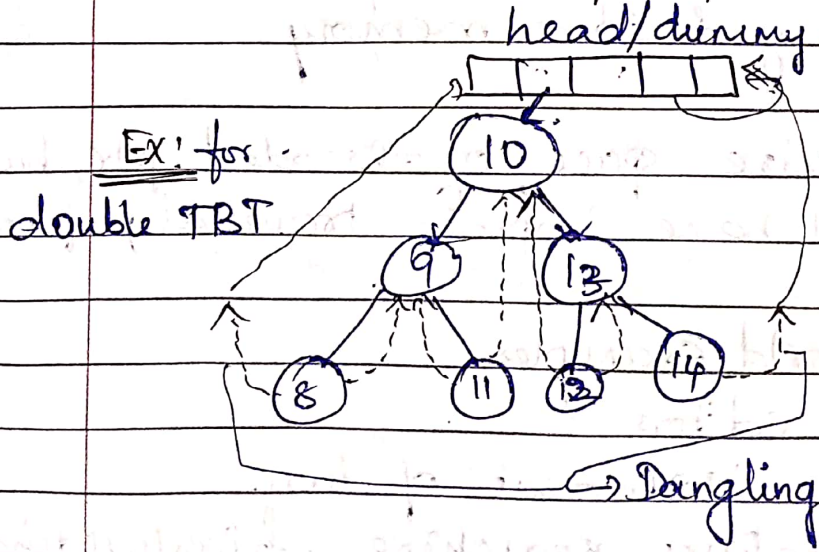
5) data searching & accessing compressed text.

Node structure of TBT.



lbit – 0 (means thread) → when it holds adr of parent node/ancestor.
– 1 (means ptr present)
↳ when left ptr holds the adr of child node

rbit – 0 (means thread)
– 1 (means ptr present)



Ex: for double TBT

Inorder traversal.

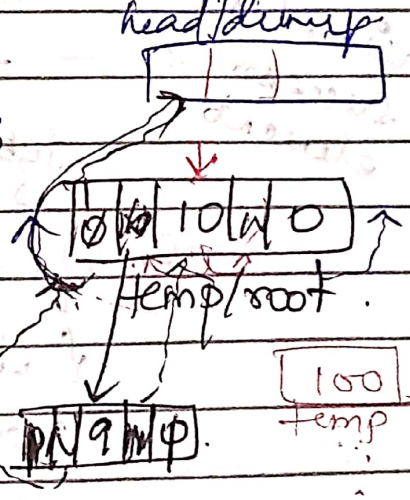
8 9 11 10 12 13 14

Implementation of TBT

```
Struct node
{
Struct node *left, *right;
int data;
int lbit, rbit;
};

Create()
{
do{
  temp = newnode;
  // Get i/p from the user. for data
     ↳temp → data;
  temp → left = NULL;
  temp → right = NULL;
  temp → lbit = 0;
  temp → rbit = 0;

if(root == NULL)
   { root = temp;
     head = newnode;
     head → data = 999/x;
     head → right = head;
     head → rbit = 1;
     head → left = root;
     head → lbit = 1;
```

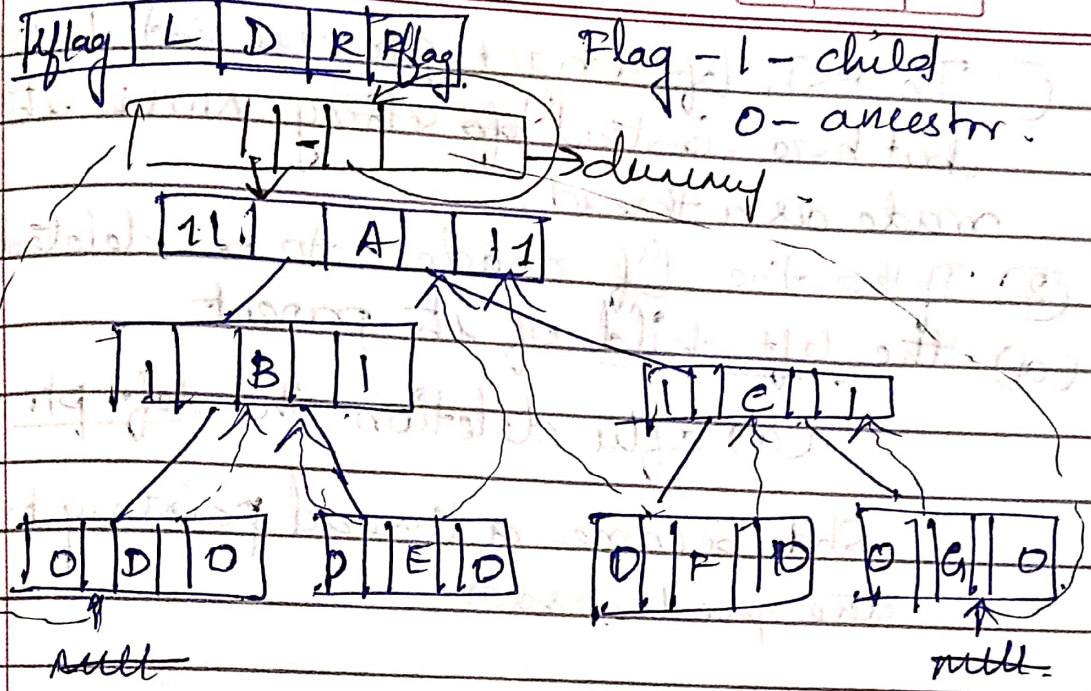


```
root->left = head;
root->right = head;
}
else temp = newnode;
{ insert(root, temp);
  // do you wanna continue (y/n)
  }while(choice == 'y');
}
```

Insertion:

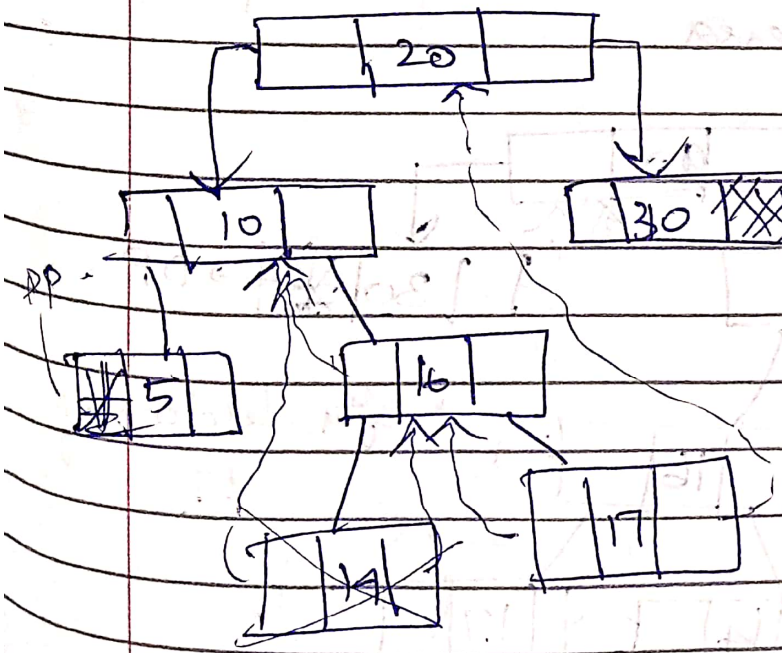
```
insert(node *root, node *temp)
{ if(temp->data < root->data)
    { if(root->left == NULL) -> if 1st element.
      root->left = temp;
      temp->left = root->left;
      temp->right = root;
      root->lbit = 1;
    } else insert(root->left, temp);
else if(temp->data > root->data)
    { if(root->right == NULL)
      root->right = temp;
      temp->left = root;
      temp->right = root->right;
      root->rbit = 1
    } else insert(root->right, temp);
}
```

inorder successor element



Deletion in TBT

case 1: leaf node to be deleted.



inorder traversal: 5 10 14 16 17 20 30.

deletion node: 14.

inorder traversal: 5 10 16 17 20 30;

PAGE NO.:

① In BST, left & right ptr are set to NULL but here instead of setting NULL its made as a thread.

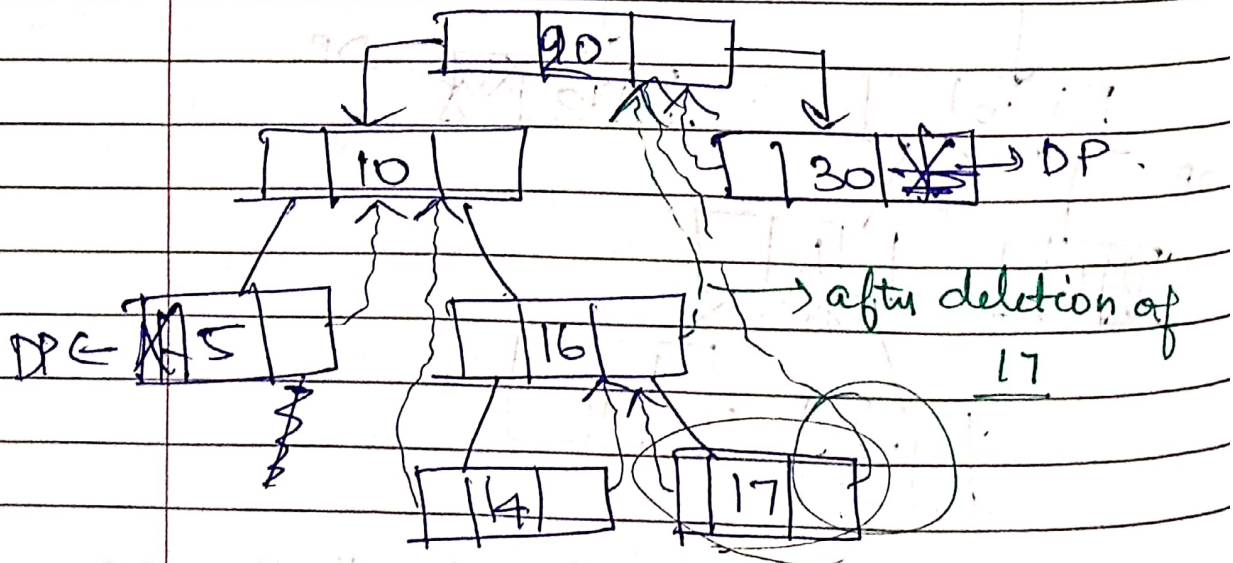
② If the the left node to be deleted is

(a) the left child of its parent

(a) after deletion the left ptr shd become a thread pointing to the predecessor.

(b) right child of its parent.

(b) after deletion, the right ptr shd become a thread pointing to the successor.



Inorder traversal: 5 10 14 16 17 20 30

deletion node: 17

after Inorder traversal: 5 10 14 16 20 30.

PAGE NO.:

Deletion

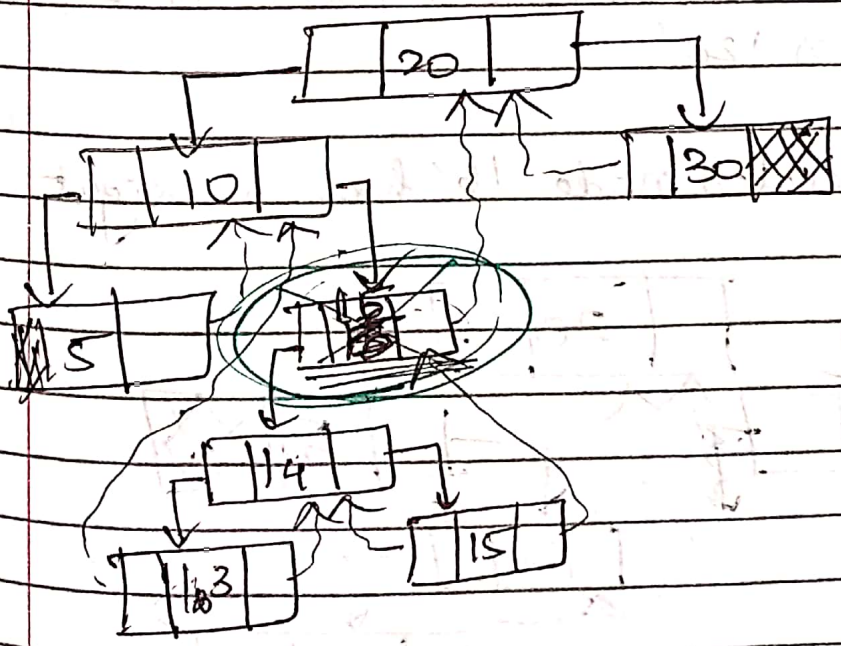
Case B: Node to be deleted has only one child.

Algo:

① After the deletion of node in BST, the inorder successor/predecessor of the node are found out.

```
s = in_successor(ptr);
p = in_predecessor(ptr);
```

② If the node deleted has a left subtree then after deletion right thread of its predecessor should point to the successor.



Inorder traversal: 5 10 13 14 15 16 20 30

Deletion node: 16.

After deletion Inorder traversal: 5 10 13 14 15 20 30

(a) Before deletion 15 is successor & 20 is predecessor of 16.

after deletion of 16, 20 becomes the successor of 15., right thread will point to 20.

Steps:

(1) delete the node (16).

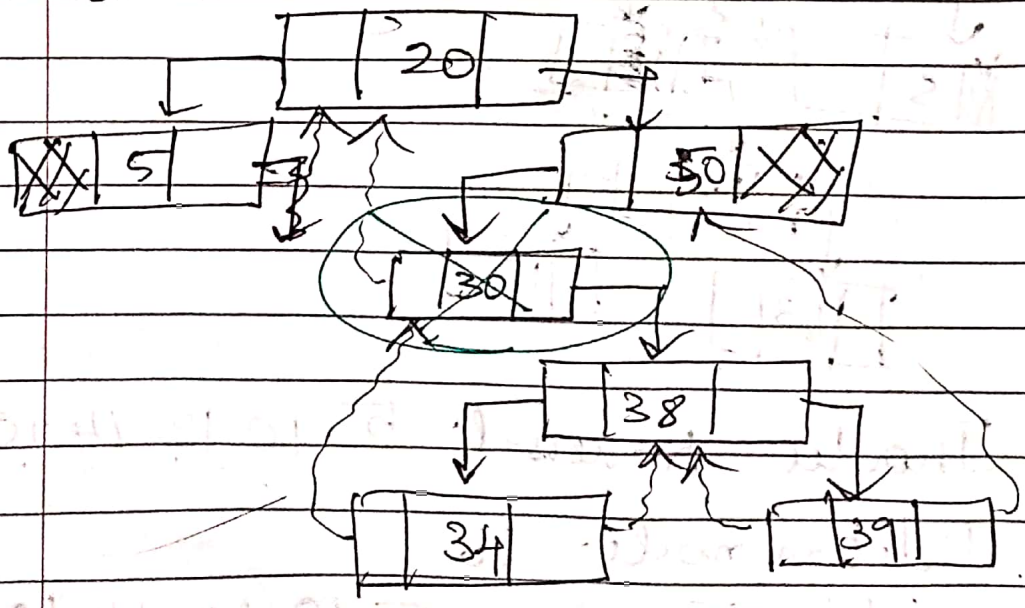
(2) Replace the left child in that place.

(3) Set the thread ptr.

Here, the deleted node '16' has a left subtree, Nah 14.

After deletion of 16, right thread of predecessor '15' should point to its successor '20'

(b) If the deleted node '16' has a right subtree.

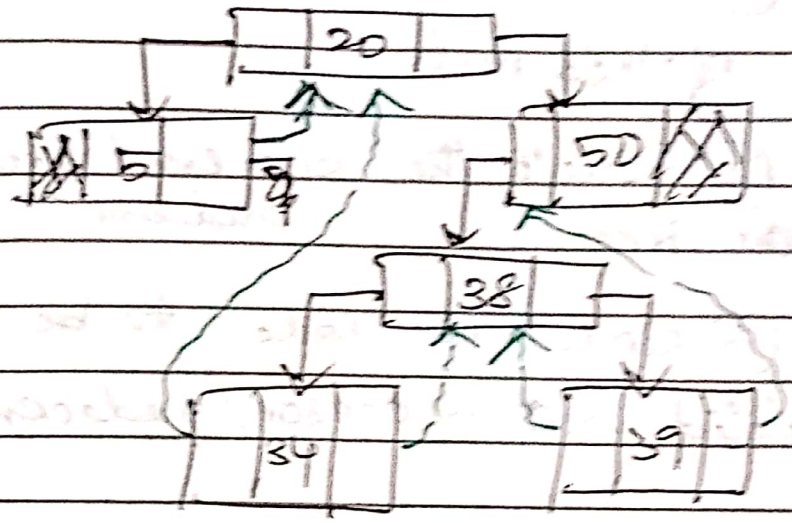


Inorder traversal: 5 20 30 34 38 39 50

deleted node: 30.

After deletion Inorder traversal

5 20 34 38 39 50.



* Here, the deleted node 34 has a predecessor 20 & successor 34

After the deletion of 30, the left thread of successor 34 points to the predecessor '20'.

Case 2: node to deleted has two children.

(1) Suppose Node N to be deleted.

(2) Find the inorder successor of node N

(a) copy the info of inorder successor to node N

(3) After deleting a node by using inorder successor.

a (or) case b

PAGE NO.:

Node with 2 children.

(1) Find the successor/predecessor of this node

(2) Delete the successor from the tree

(3) Replace this node to be deleted with successor/predecessor.

Unit-4

AVL tree (Georgy Adelson-Velsky & Evgenii Landis)

- Its a BST.
- Balance factor = |height of left subtree| - |height of right subtree|
  (-1, 0, 1).
- BST → no duplicates is allowed.
- its a self balancing BST.
- The tree will be balanced for every insertion & deletion.

Operations:

- insertion
- deletion
- searching [similar to BST].

Uses:

1. Data retrieval.
2. DB → to keep track of files ly.
3. Data compression
4. Text editors → search the data & replace.

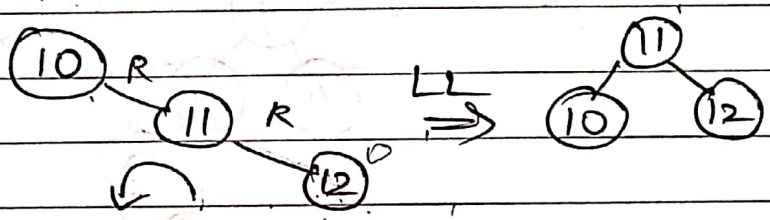
Why?

Balanced
- to maintain the BST.
  - this improve the efficiency, consistent & predictable time complexity for search, insert & delete operations.

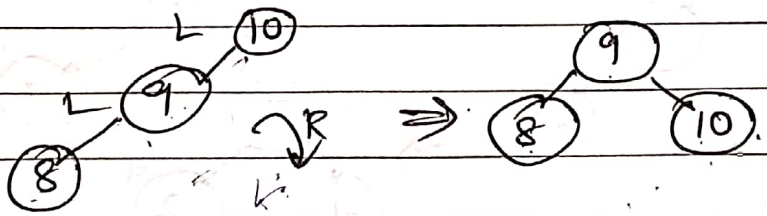
4 cases:

1. LL Rotation
2. RR Rotation
3. LR Rotation
4. RL Rotation.

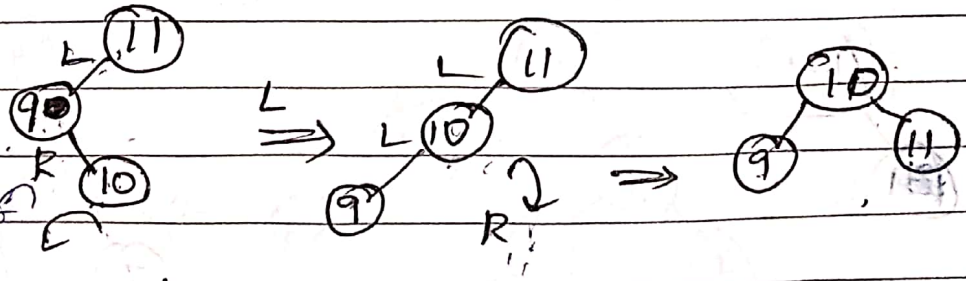
1. ~~RR~~ LL Rotation



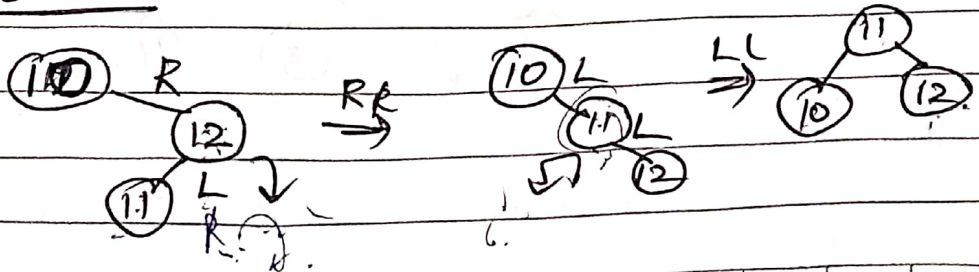
2. RR Rotation



3. LR Rotation

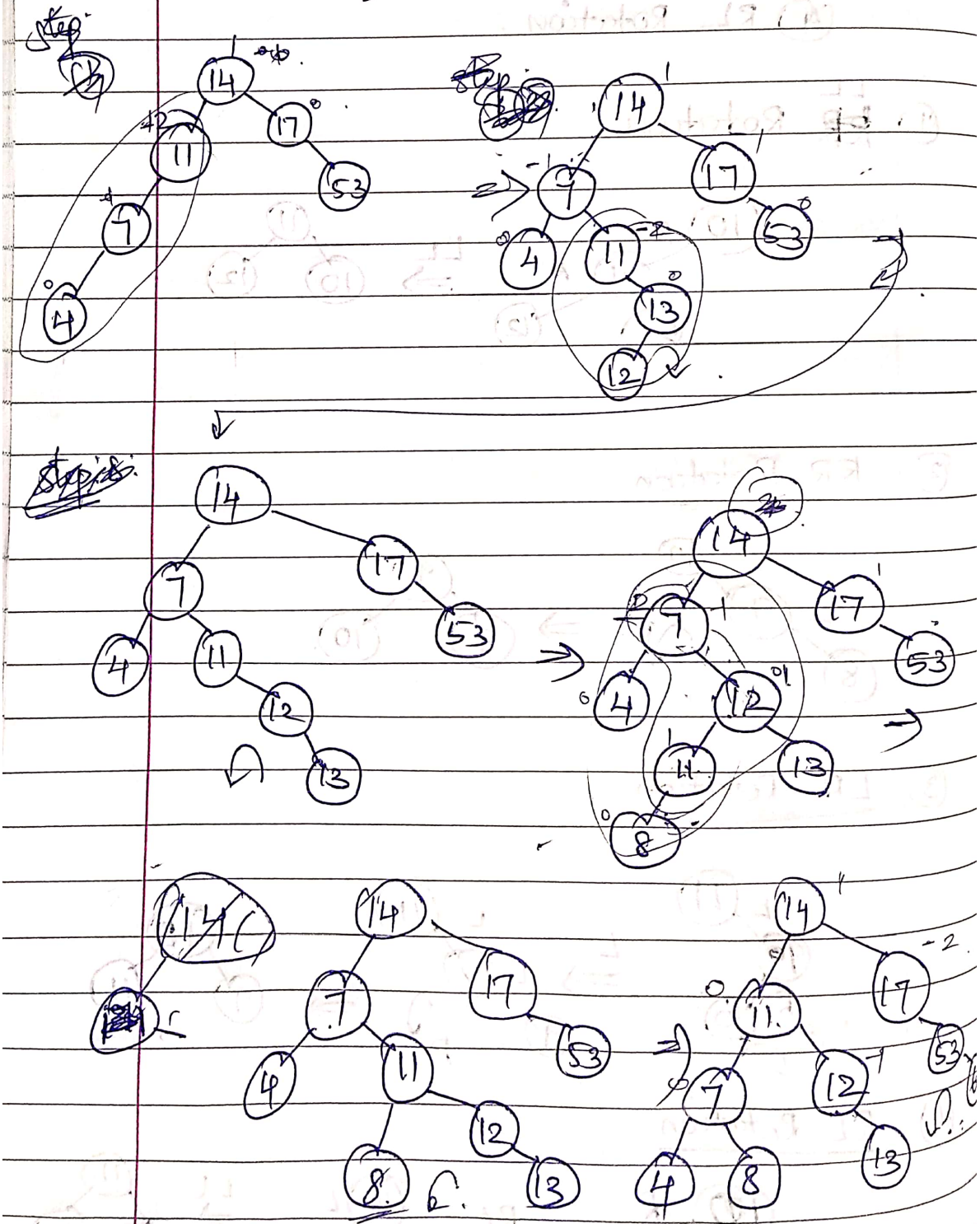


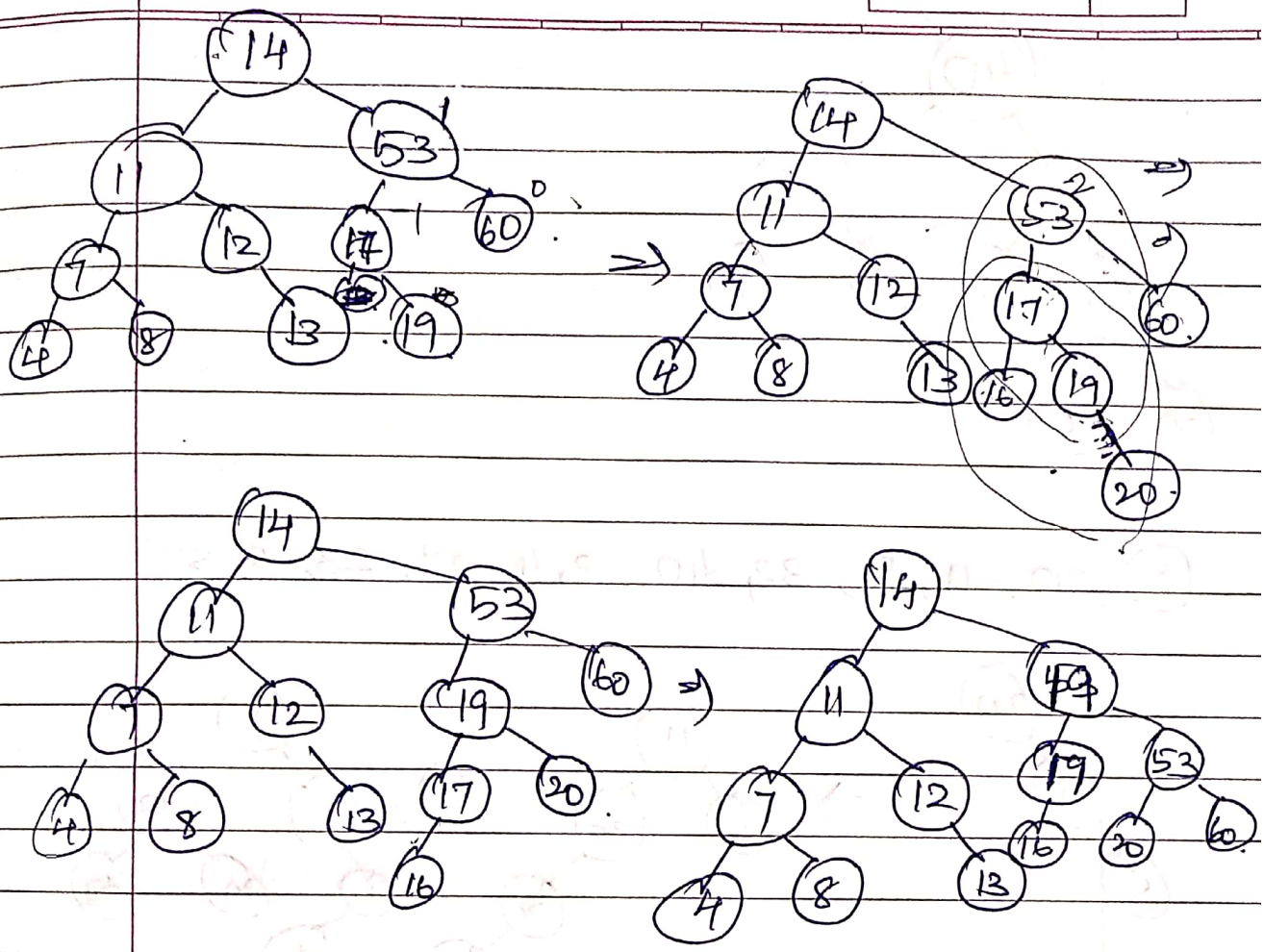
4. RL Rotation



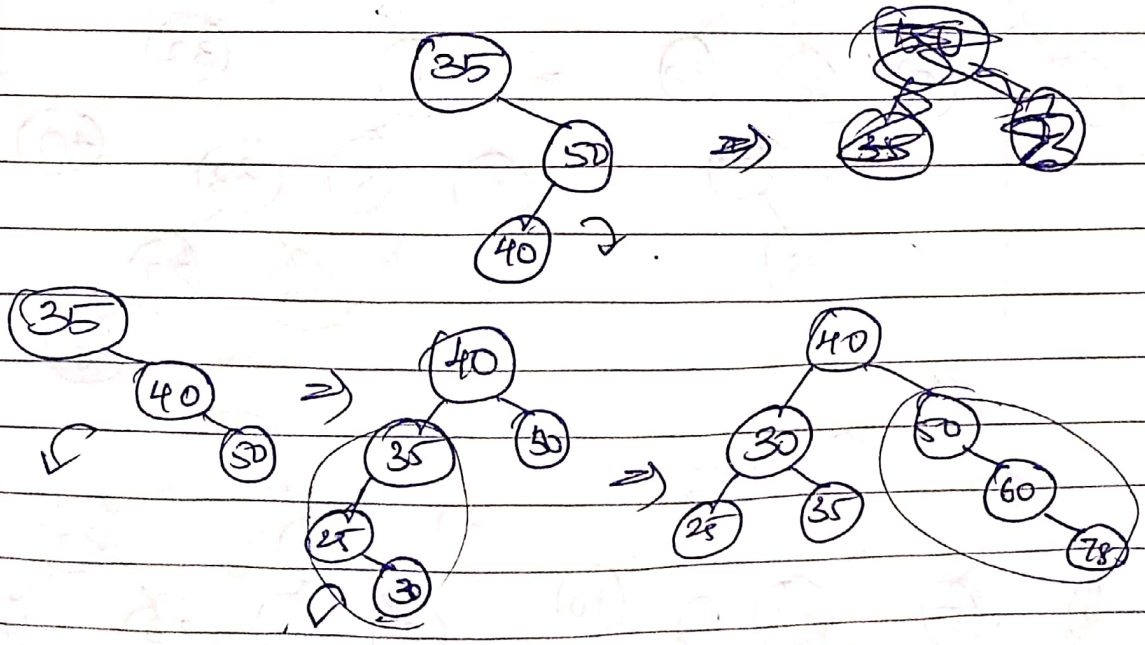
Construct an AVL tree by inserting the following data.

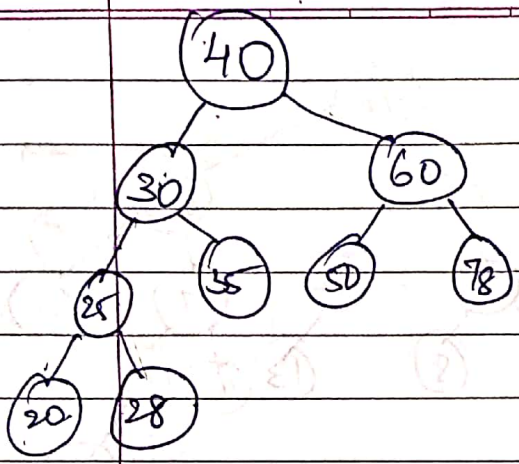
14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20.



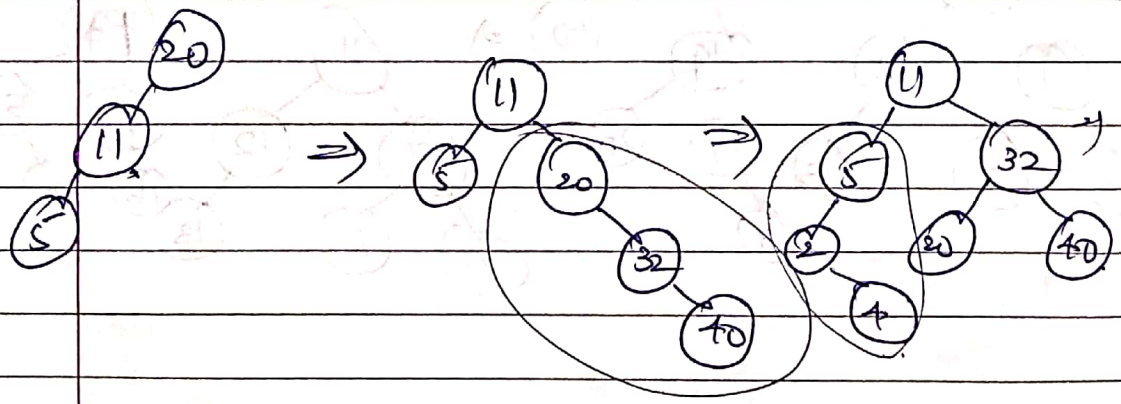


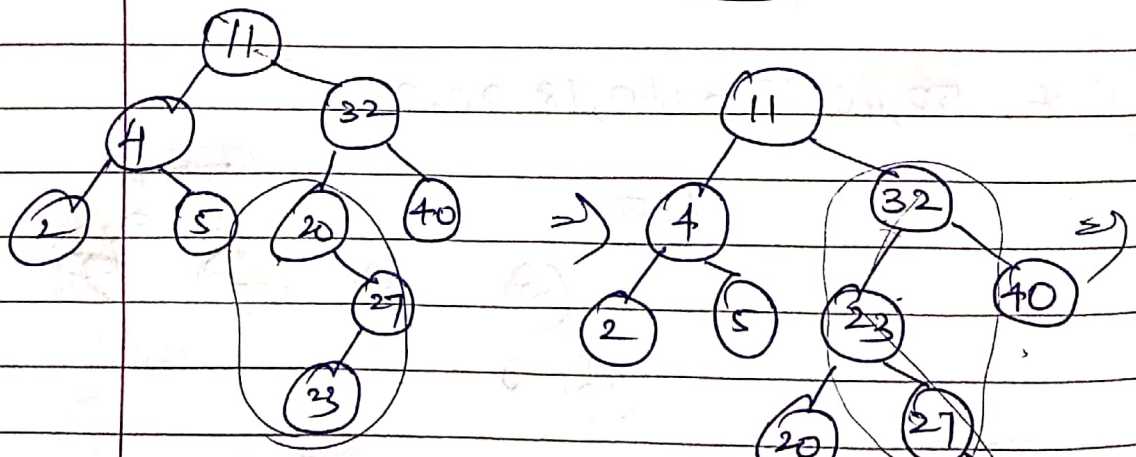
(2) 35, 50, 40, 25, 30, 60, 78, 20, 28.

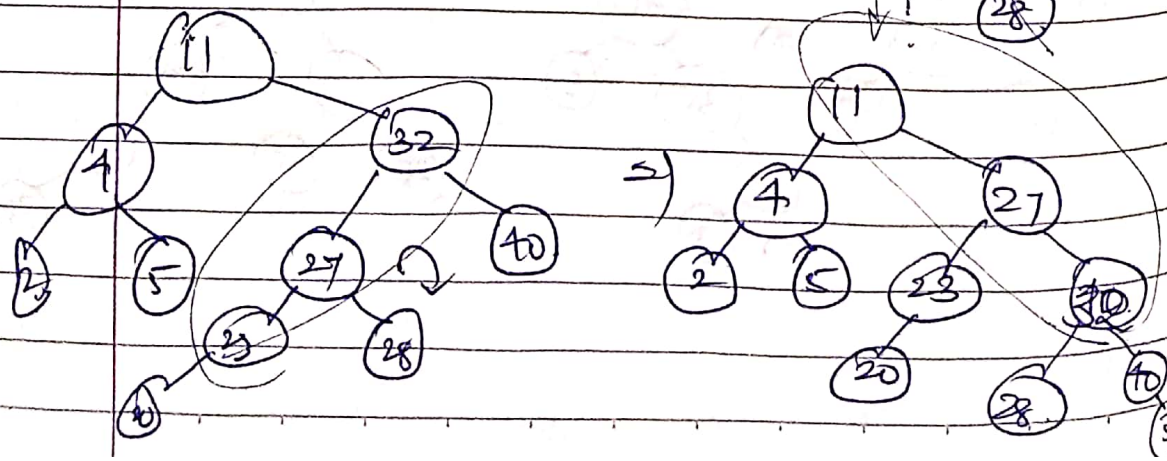


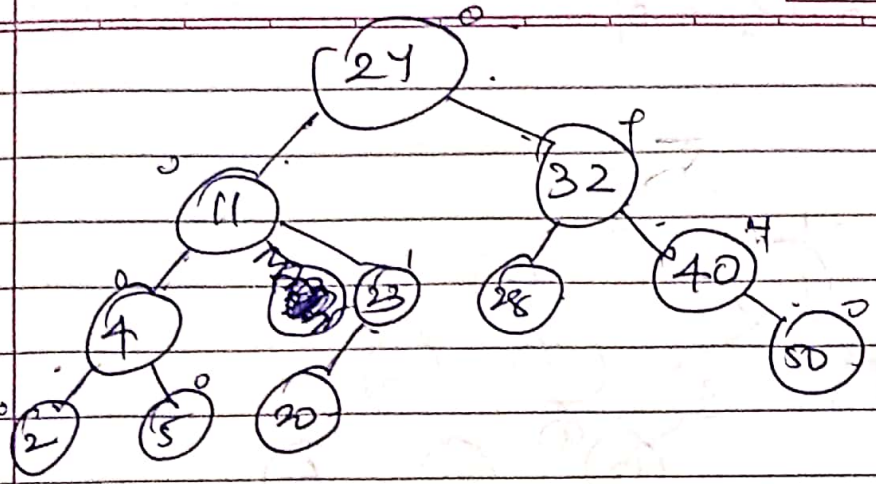


(3) 20, 11, 5, 32, 40, 2, 4, 27, 23, 28, 50.

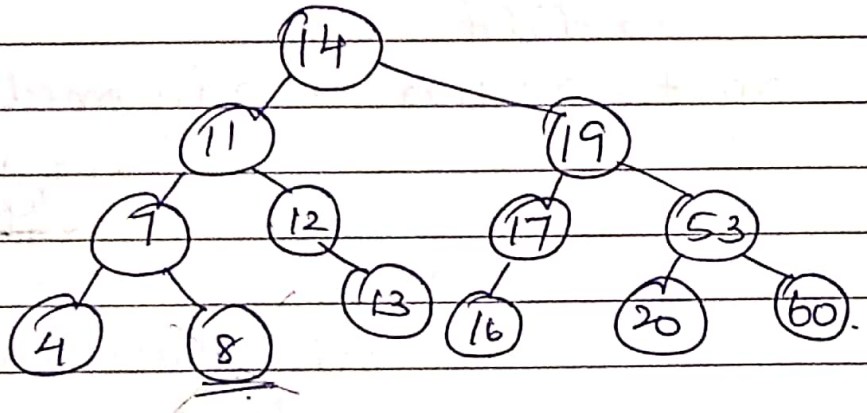








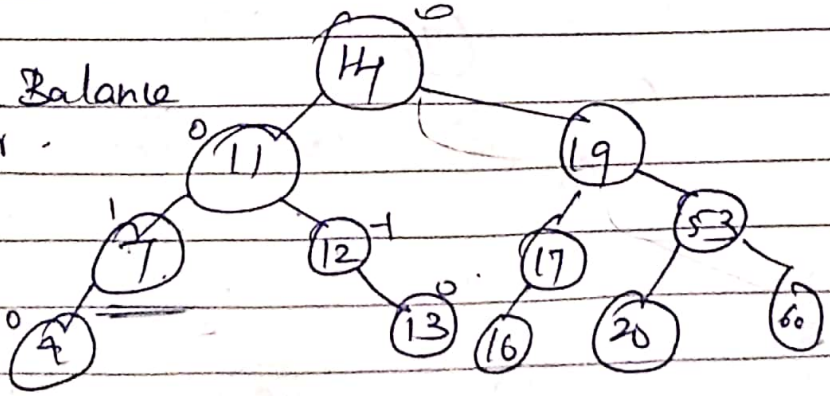
Deletion in AVL tree: (same as BST — Child 0, 1, 2)



(1) To delete the following data
8, 7, 11, 14, 17.

(1) after (delete).

(2) check Balance factor.

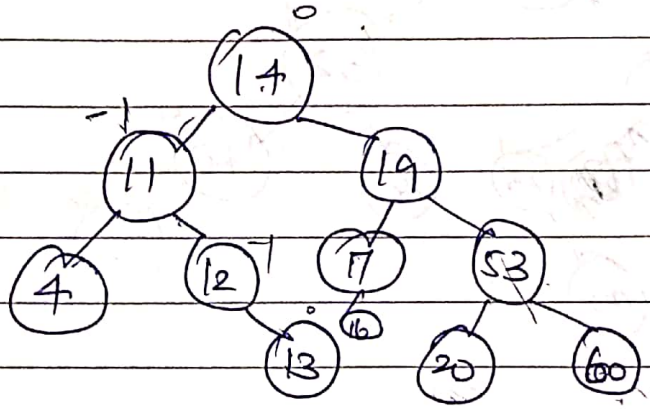


M T W T F S S

Page No.: YOUVA

Date:

after deletion of (7).



after deletion of (11).

↳ 2 child.

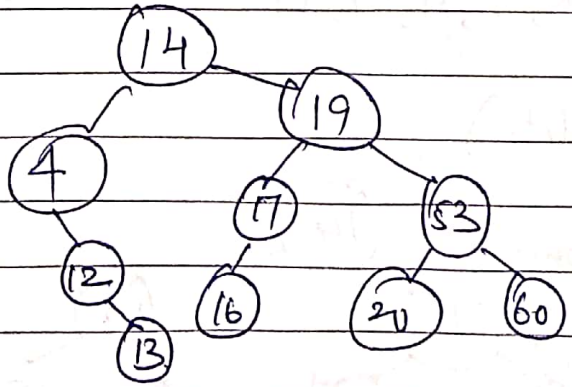
root → either inorder predecessor.

↳ (largest element in the ~~right~~ left subtree)

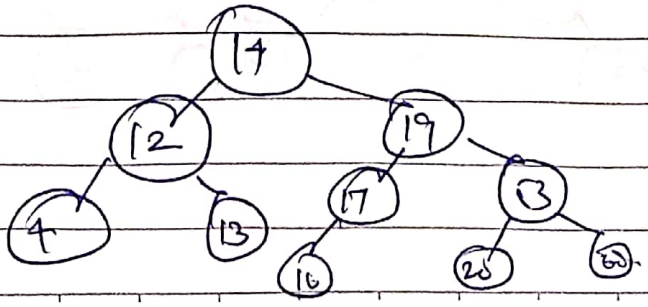
(or)

inorder successor.

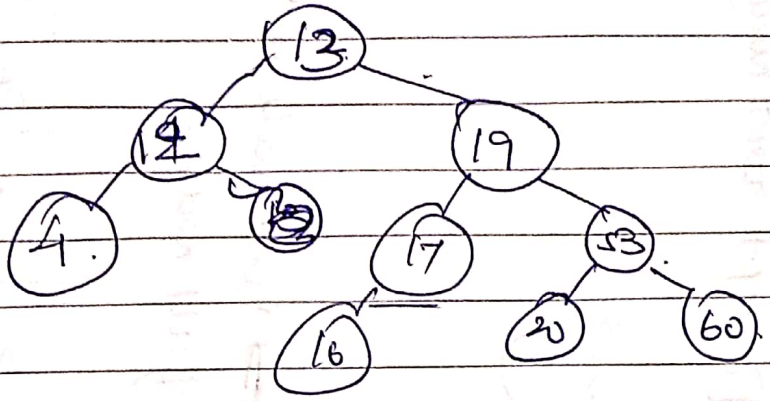
↳ (Smallest element in the right subtree).



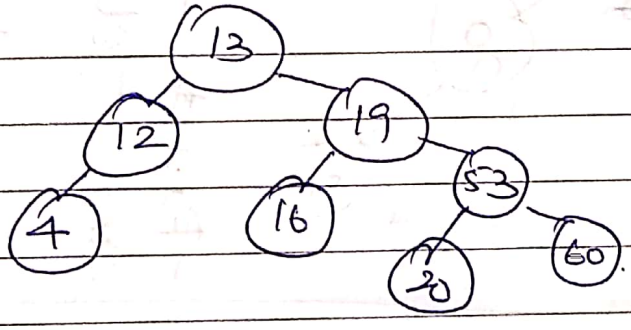
⇒



after deleting of (14).



after deleting (14).



Heap: (Insertion, Deletion, heapsort, heapify, array rep., priority queue)

→ tree based Data structure

→ almost complete Binary tree

↳ means: all the levels are completely filled except the last level. In the last level all the keys are aligned to the left (as left as possible)

⇒ follow a ACBT (almost Complete Binary tree)

Types of heap

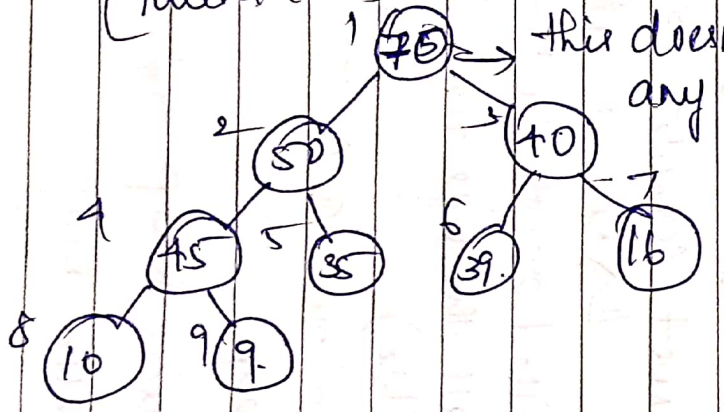
↳ max heap

↳ min heap.

Max Heap

for every node 'i', the value of node is $\leq$ to its parent node (descendent value).

[Parent(i)] $\geq$ A[i] [except root node]



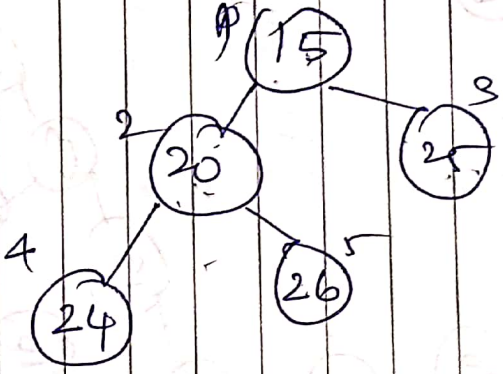
this doesn't have any parent.

	70	50	40	45	25	39	16	10	9	
A	1	2	3	4	5	6	7	8	9	

root node will ~~also~~ always be greater than all other node.

Min Heap.

for every node i, the value of node is $\geq$ to its parent node [except root node]



~~Parent node need~~ Root node is the smallest element of all nodes.

	15	20	25	24	26	
A	1	2	3	4	5	

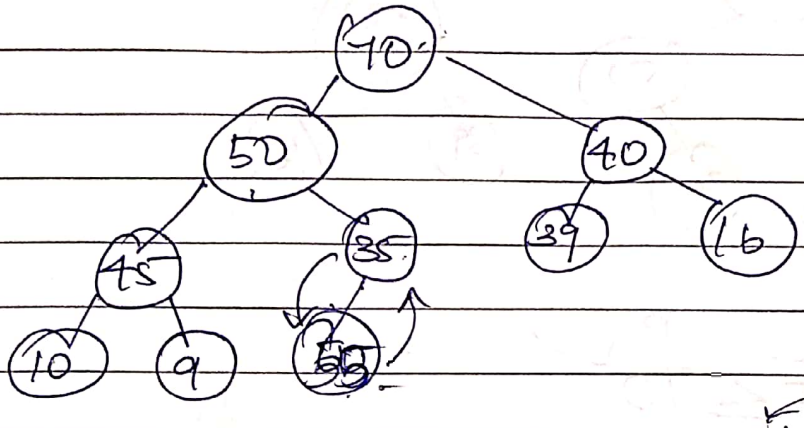
Max Heap → insertion → O(log n)

Max.

M T W T F S S
Page No.:
Date:
YOUVA

↳ insert the element from leaf node not root node.

insert the element: 55.



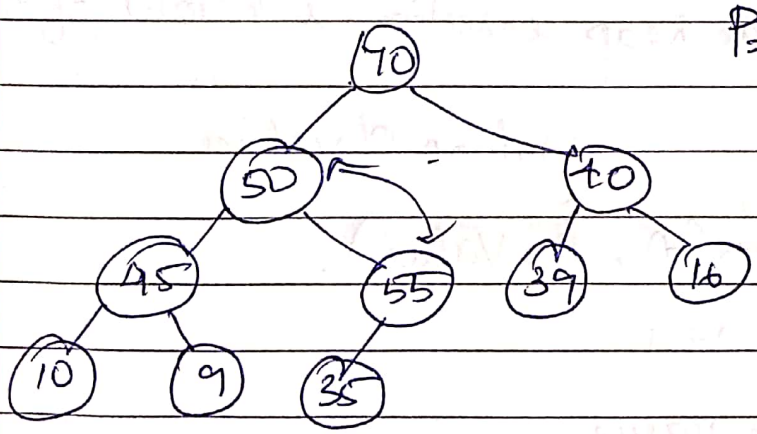
A | 70 | 50 | 40 | 45 | 35 | 39 | 16 | 40 | 9 | 55 |
1 2 3 4 5 6 7 8 9 10.

55 ≥ 35

∴ P ⟹ swap.

$P = \left\lfloor \frac{i}{2} \right\rfloor = \frac{10}{2} = 5$.

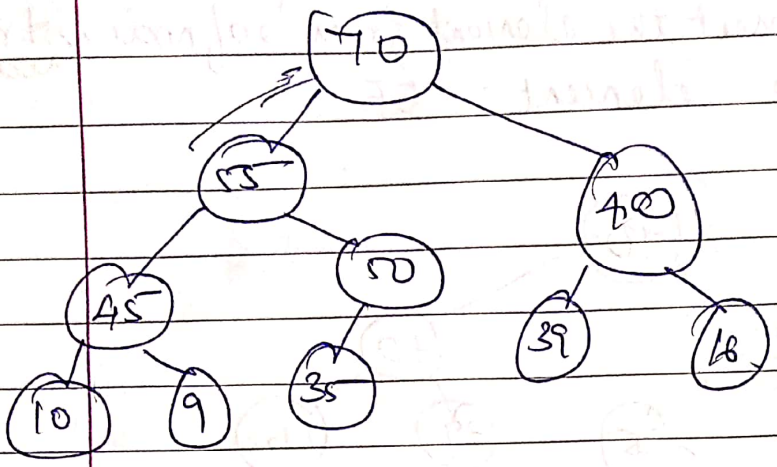
↳ floor.



A | 70 | 50 | 40 | 45 | 55 | 39 | 16 | 10 | 9 | 35 |

Parent ≤ child

50 ≤ 55. ⇒ swap.

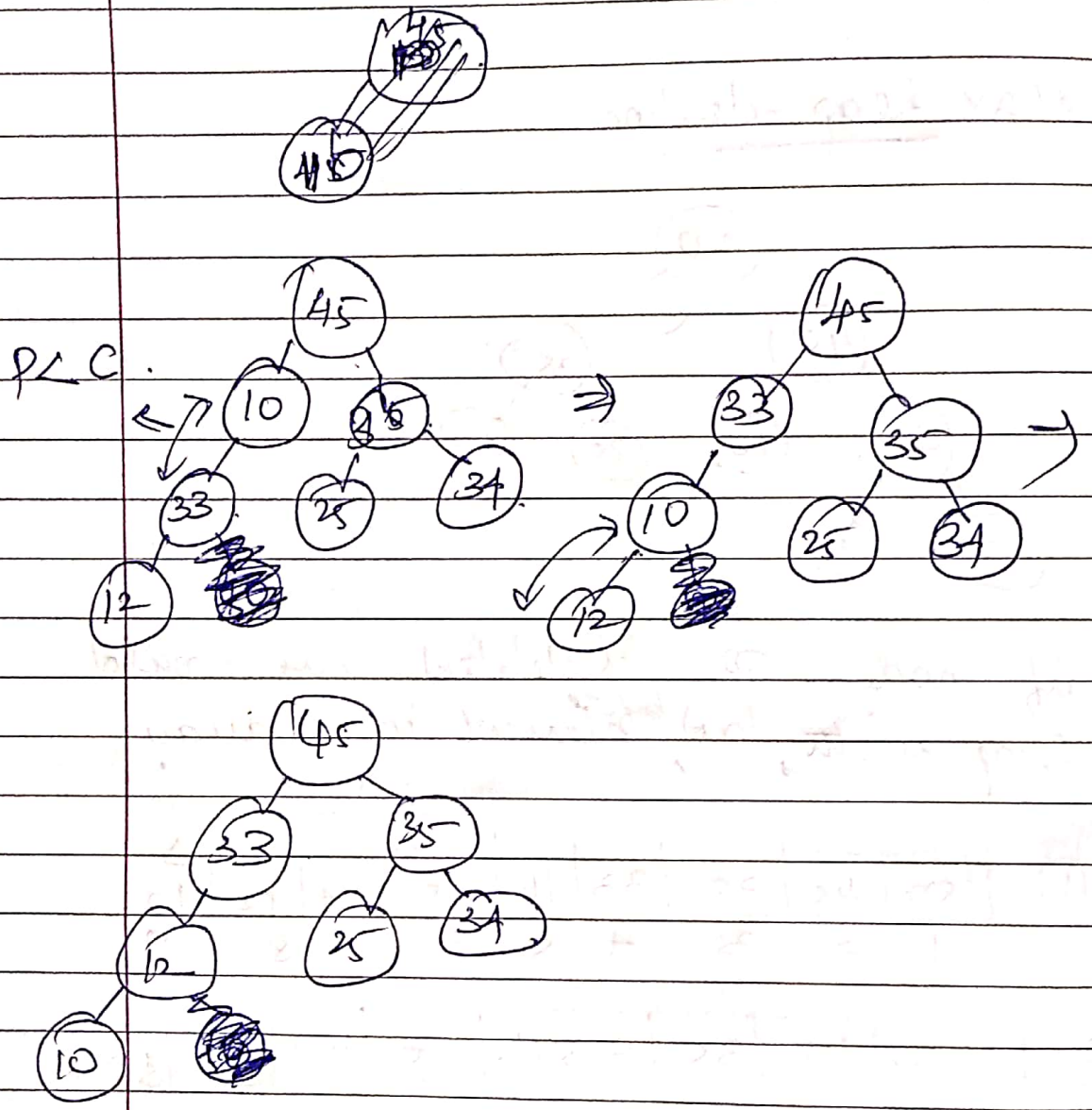


C P
55 ≤ 70 ✓

A	70	55	40	45	50	39	16	10	9	35

Note: ~~height of max~~ max. no. of comparisons in max heap insertion is height of the tree.

```
                    ~total no. of values
                    ↑
insertHeap(A, n, value).
{ n=n+1
A[n] = value;
i = n;
while(i>1)
{parent = floor(i/2);
if(A[parent] < A[i])
{
swap(A[parent], A[i]);
i=parent;
}
else
```



max heap:

insertion : the tree is adjusted from leaf to root.

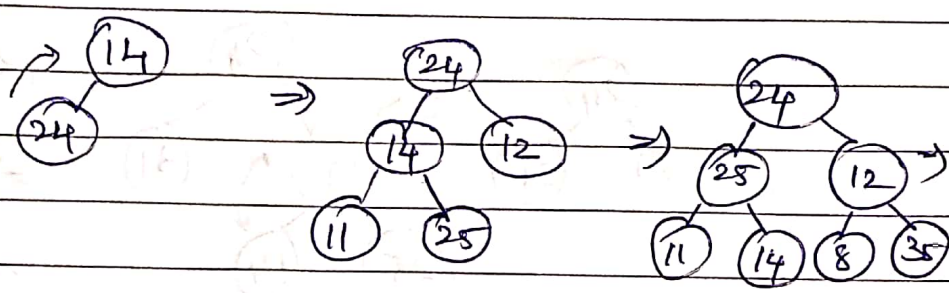
deletion : the tree is adjusted from root to leaf.

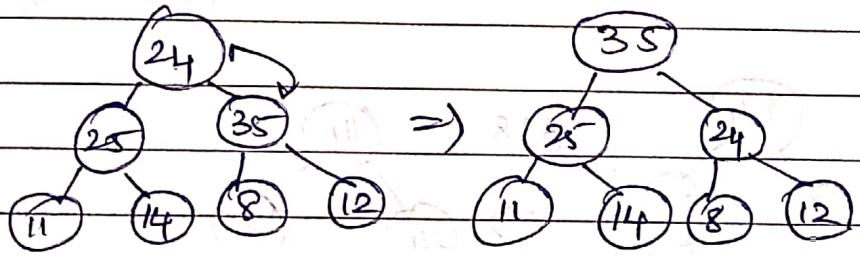
left child = (2*i)

right child (2*i)+1

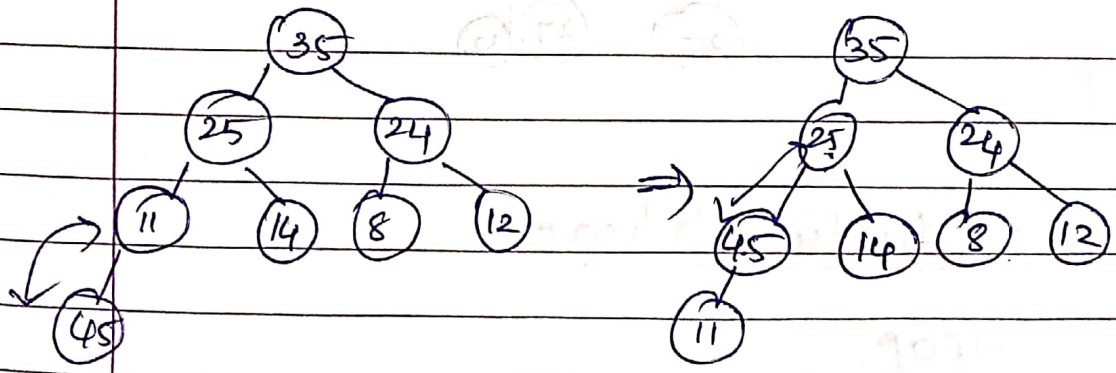
Max heap - creation.

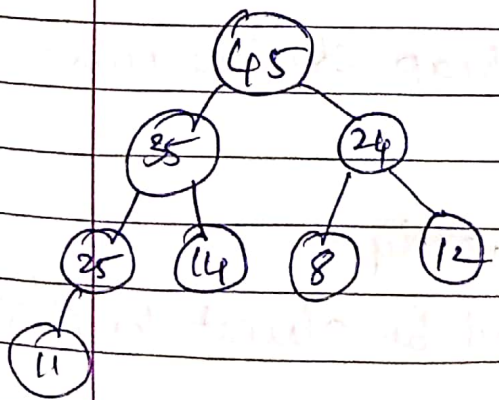
14, 24, 12, 11, 25, 8, 35





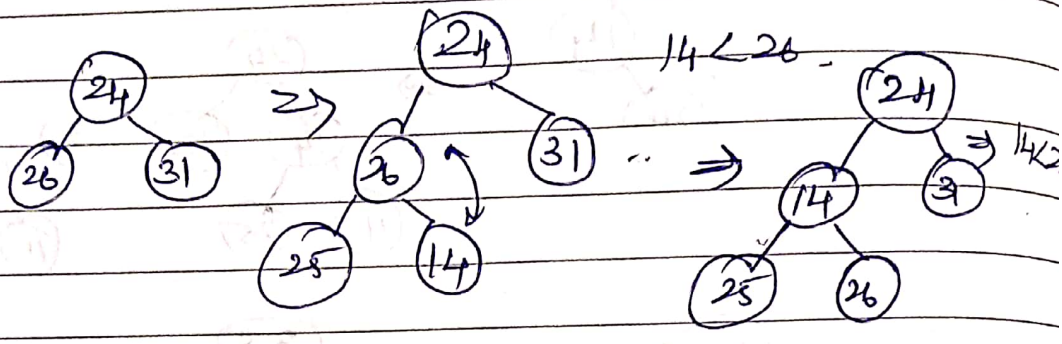
insert 45.

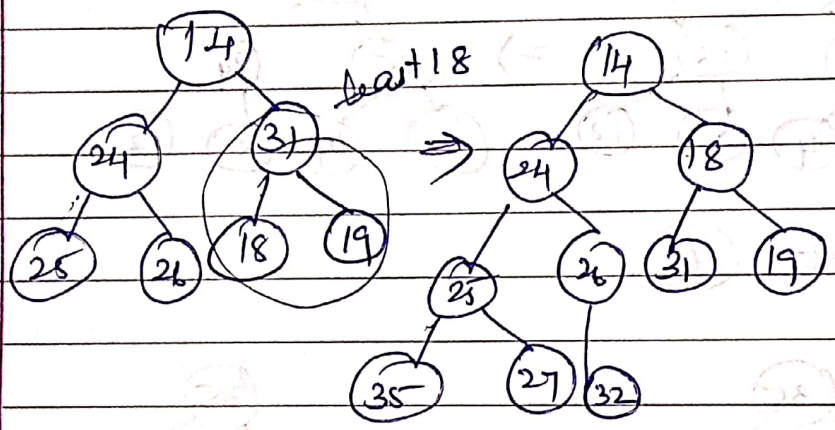




Minheap - creation

24, 26, 31, 25, 14, 18, 29, 35, 27, 32





Introduction to heap.

Heap:

When creating a heap structure must follow 2 property:

(1) structural property.

↳ it shd be almost complete Binary tree (ACBT)

(2) Ordering property.

→ Max heap (Parent > child)
→ Min heap (Parent < child)

M T W T F S S
Page No.:
Date:
YOUVA

• Mostly max-heap is preferred.

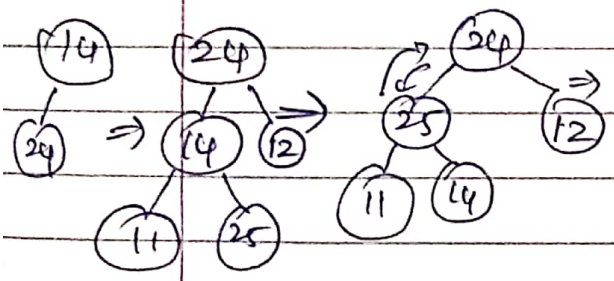
Heap tree construction

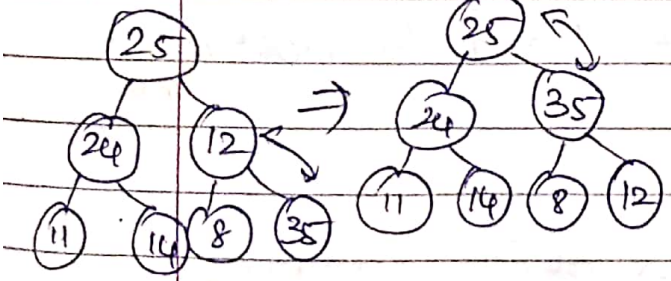
Insert key one by one in the given order.

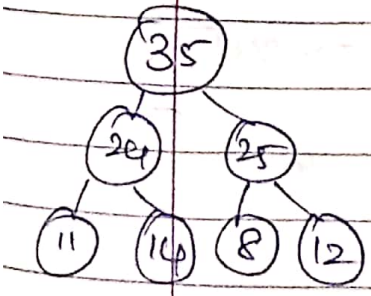
$O(n \log n)$

Max heap

14 24 12 11 25 8 35







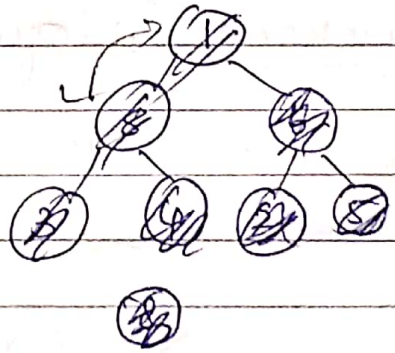
Build heap / Heapify method

$O(n)$

n = heap size.

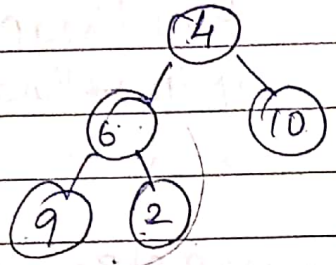
Range of leaves $\left\lfloor \frac{N}{2} \right\rfloor$

(1, 5, 6, 8, 12, 14, 16)
IN ... LN ..

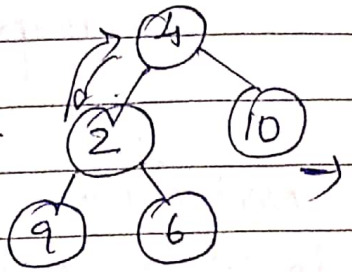


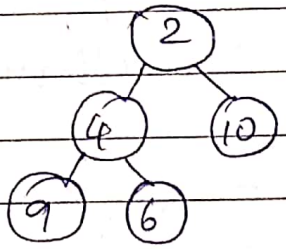
Heap sort (heapify → built-in method)

4 6 10 9 2



Min heap.



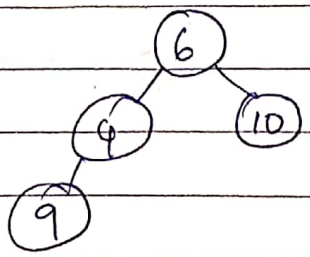


O(n) ⇒ time complexity.

Deletion: (n log n).

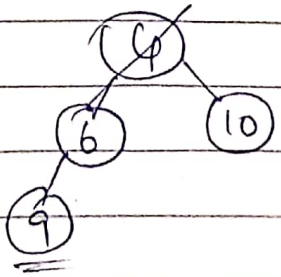
Step 1: delete the root. (Bring the eight most element to the root)

2.

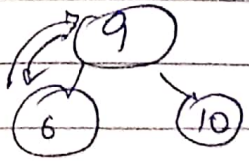


⇒ not following min heap property.

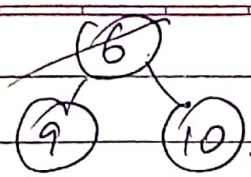
2 4.



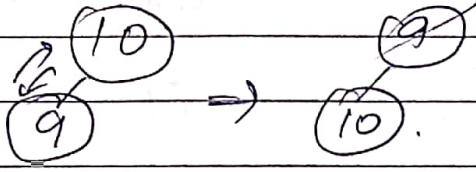
Step 2:



⇒ heapify.

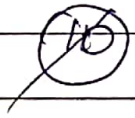


Step3: 2 4 6



2 4 6 9.

Step4:



2 4 6 9 10.

Steps:

(1) First create a the heap.

(2) Then change to min-heap if you want (a) given sequence of elements to be arranged in ascending order.

(b) Max heap if you want given sequence of elements to be arranged in descending order

(3) Then perform the deletion operation to get the sorted list.

⇒ In heap, deletion is performed by deleting the root of the tree and replacing it with rightmost element.

Creation of heap:

2 methods

- one by inserting one by one / single element checking whether it satisfy the min-heap / max-heap property. (nlogn).
- Using heapify / built heap method. time complexity O(n).

Heap sort: (by default unstable)

Inplace

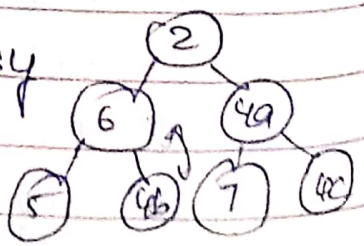
⇒ it works where the tree is created, it does not take any extra space.

Unstable:

a b c

2 6 4a 5 4b 7 4c

⇒ positions changes that's y unstable.

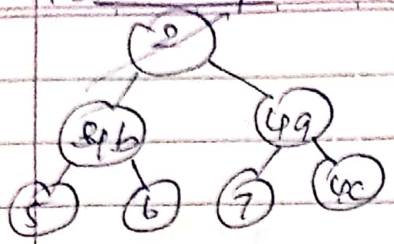


M T W T F S S

Page No.: YOUVA

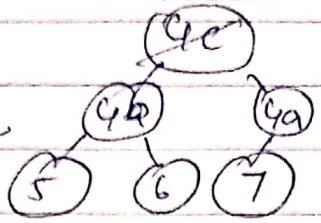
Date:

Min heap

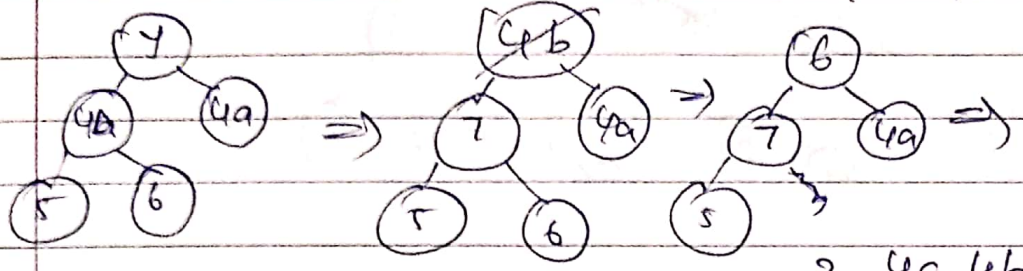


Heap sort:

2 4c

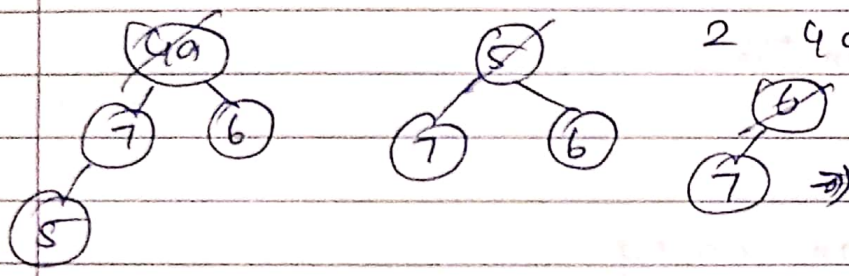


2 4c 4b



2 4c 4b 4a

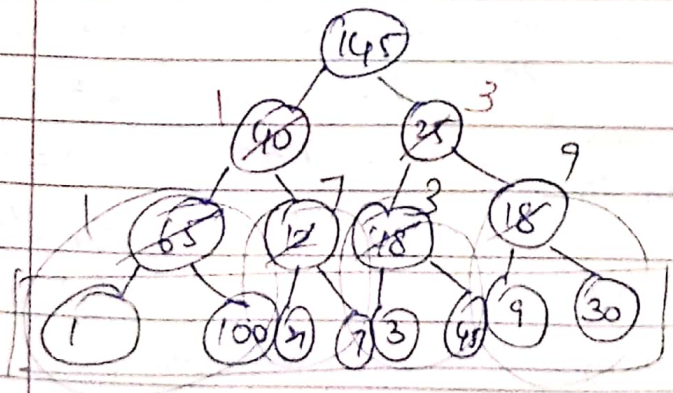
2 4c 4b 4a 5

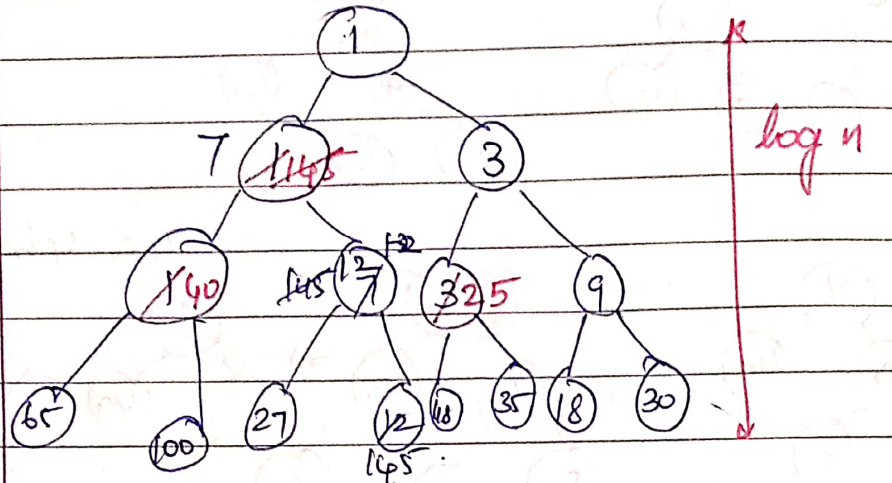


2 4c 4b 4a 5 6 7

c b a

Build-heap:



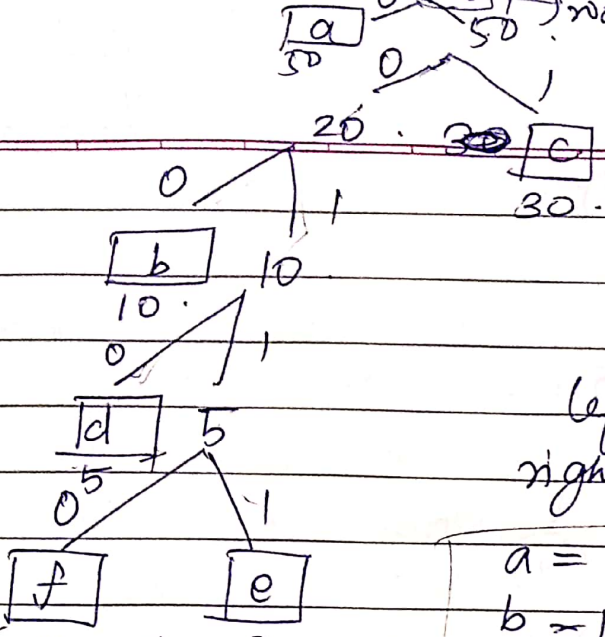


Huffman tree:

- Huffman coding
  - → greedy technique ~~is used~~.
  - → it used for encode, data compression.

a = 50		a = 000	for representing each character 3 bit is used
b = 10, c = 3, d = 2.	5, 10, 20	b = 001	
c = 30		c = 010	
d = 5		d = 011	
N = 100 characters		e = 100	3 x 100 = 300.
		f = 101.	then only the given data will be encoded.

ASCII (0-127)
8 x 100 = 600 bit.



element:
left-small
right large.

left ⇒ 0.
right → 1

a = 0.	d = 1010.
b = 100	e = 10111
c = 11	f = 10110.

(1) ↳ root to that character.

(2) a = 50 × 1 = 50
b = 10 × 3 = 30
c = 30 × 2 = 60
d = 5 × 4 = 20.
e = 3 × 5 = 15
f = 2 × 5 = 10.
185 ⇒ bit.

aa bbbb a bbb e ccddd eee ccc eee ddeee.
1 2 3

Ascii ⇒ a = 97.
$2^7 = 128$.

		freq. of characters.				
a = 3		3	4	4	5	9
b = 7		a	b	c	d	e
c = 6						
d = 5						
e = 9						

Min heap.