TREES II

BTree

Heaptree - complexity, comparison

Priority Queue with another Sorting Alg.

Red-Black Tree

Tries

Splay Tree

Skip list

B-Tree / Balanced m-way Search Tree

m-way search tree minimizes the time of file access due to its reduced height. But it is required to keep the height of the tree as low as possible and for this purpose we need to balance the height of the tree. The balanced m-way search tree is called B-tree.

Definition:

A B-tree of order m is an m-way search tree in which.

1. Root node has atleast 1 key value and at most $m-1$ key values.
2. The other nodes except the root contain at least $\lceil \frac{m}{2} \rceil - 1$ key values and at most $m-1$ key values.
3. All leaf nodes are on the same level.
4. Keys are arranged in a defined order within the node.

Rules for insertion of a new key in a B-tree

1. First search the tree to check the existence of new key value in the tree. If no duplicate key is found, then get the leaf node N where the

new key is to be inserted.

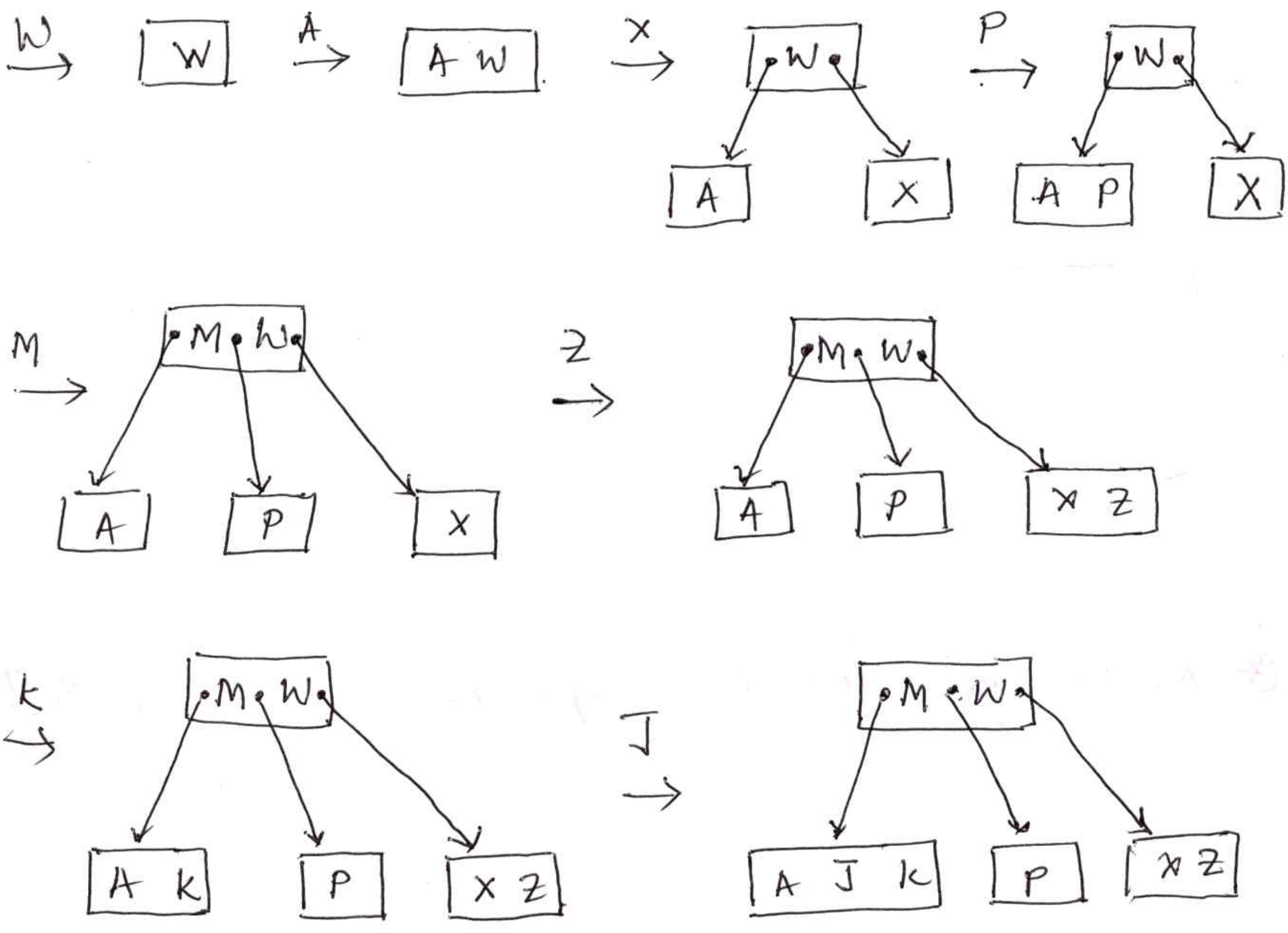
2. If N is not full, then insert the new key in N and exit.

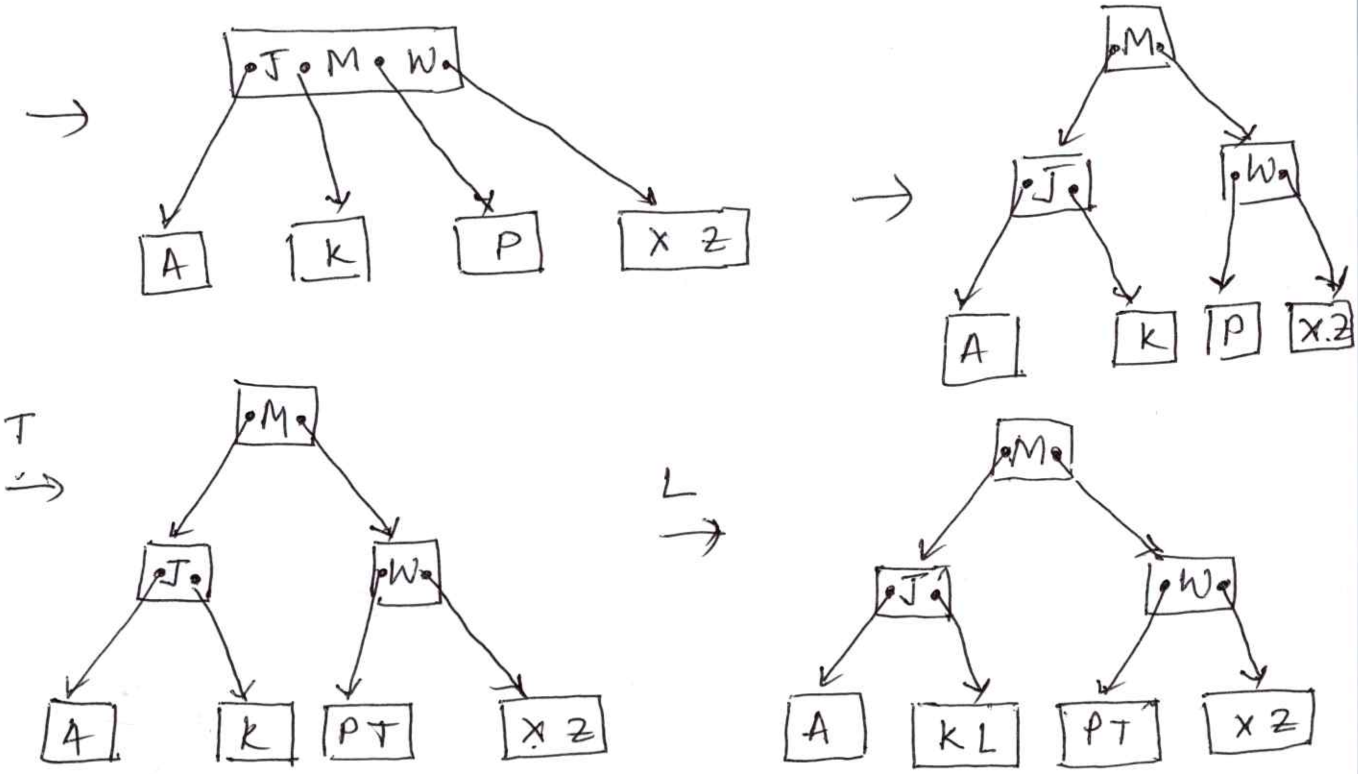
3. If N is full, then split the node into two new nodes at its median value and move the median key value to its parent node.

4. Repeat step 3 until the tree is balanced.

<u>Example</u>: (*) Insert the following elements in a B tree of order 3.

W, A, X, P, M, Z, K, J, T, L





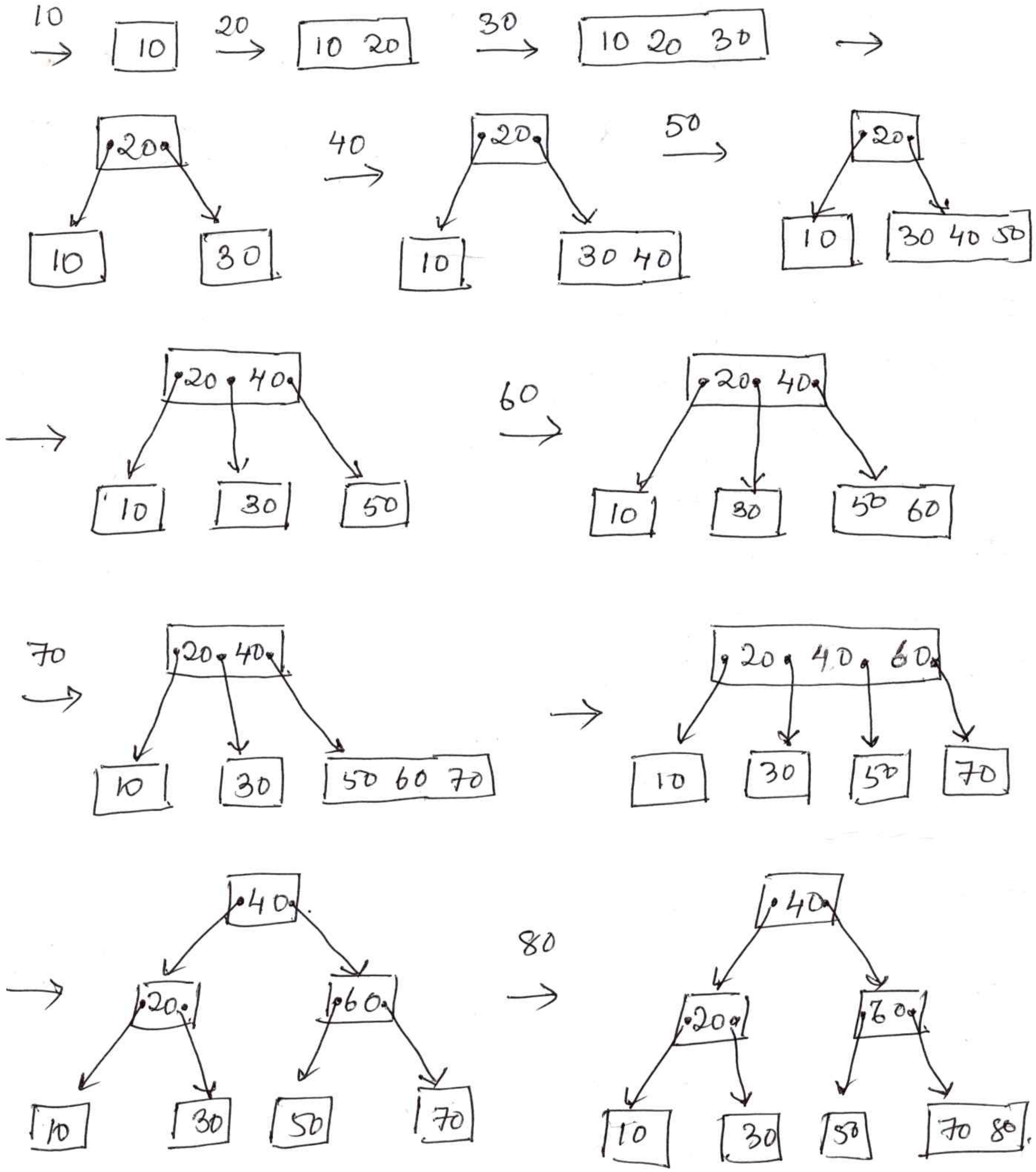
(*) When a key value is to be inserted into a node which already has the maximum number of key values. Then the following steps need to be adopted.

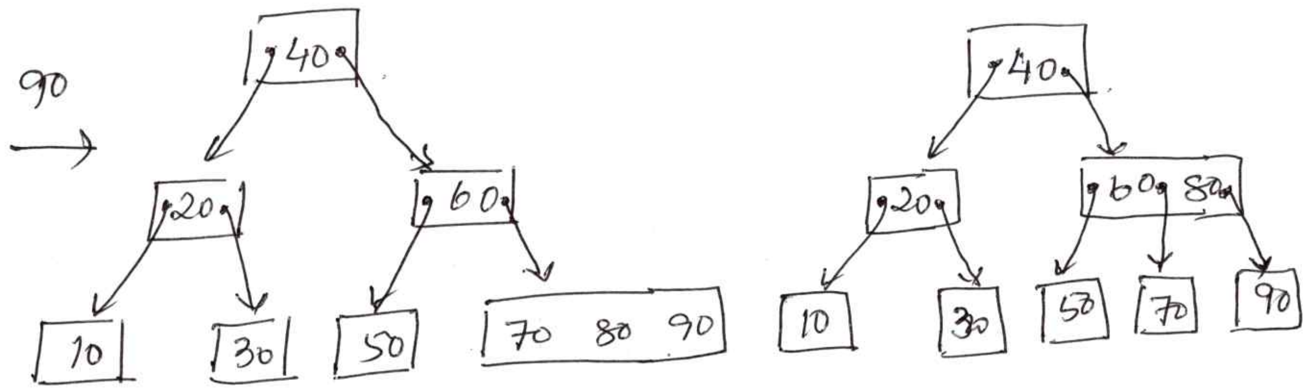
1. Insert the value into the list of values in the ascending order.
2. Split the list into three parts : P_1, P_2 and P_3
   - P_1 contains the first $\lceil \frac{m}{2} \rceil - 1$ key values.
   - P_2 contains the $\lceil \frac{m}{2} \rceil^{\text{the}}$ value.
   - P_3 contains the $\lceil \frac{m}{2} \rceil + 1, \ldots, m^{th}$ values.
3. With this splitting, the $\lceil \frac{m}{2} \rceil^{\text{the}}$ value is to be inserted into the parent node of the current node (If parent node is nil, then create a new node). In place of the current node,

two nodes are to be alloted containing the key values p_1 and p_3 respectively.

(*) Insert the following elements in a B-tree of order 3 :

10, 20, 30, 40, 50, 60, 70, 80, 90.





Heap Tree / Binary Heap

Suppose H is a complete binary tree, then H is called a heap or max heap, if for each node N of H, the value of N is greater than or equal to the value of each of the children of N.

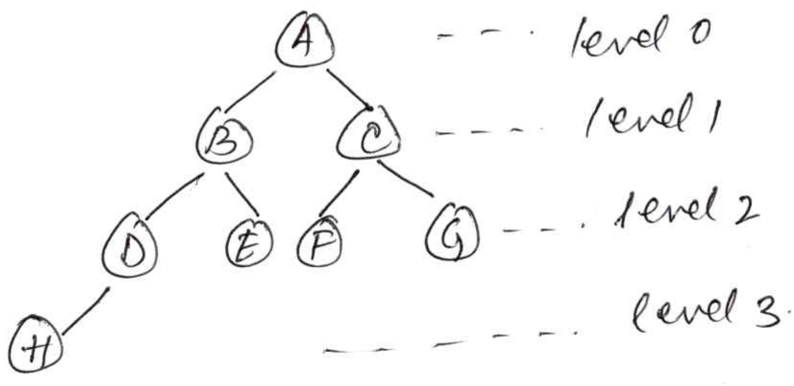
Analogously, a minheap is a complete binary tree H, where for each node N of H, the value at N is less than or equal to the value of each of the children of N.

(Unless otherwise stated, we assume that H is maintained in memory by a linear array TREE using the sequential representation of H).

In the sequential representation of a tree, the nodes are stored level by level, starting from the zero level where only the root node is present. The root node is stored in the first memory location (as the first element in the array).

This representation uses only a single linear array TREE as follows:

1. The root R of T is stored in TREE[1].
2. If a node N occupies TREE[K], then its left child is stored in TREE[2*K] and its right child is stored in TREE[2*K+1]. The root is at TREE[K/2].



1	2	3	4	5	6	7	8					
A	B	C	D	E	F	G	H					

Sequential representation of the binary tree.

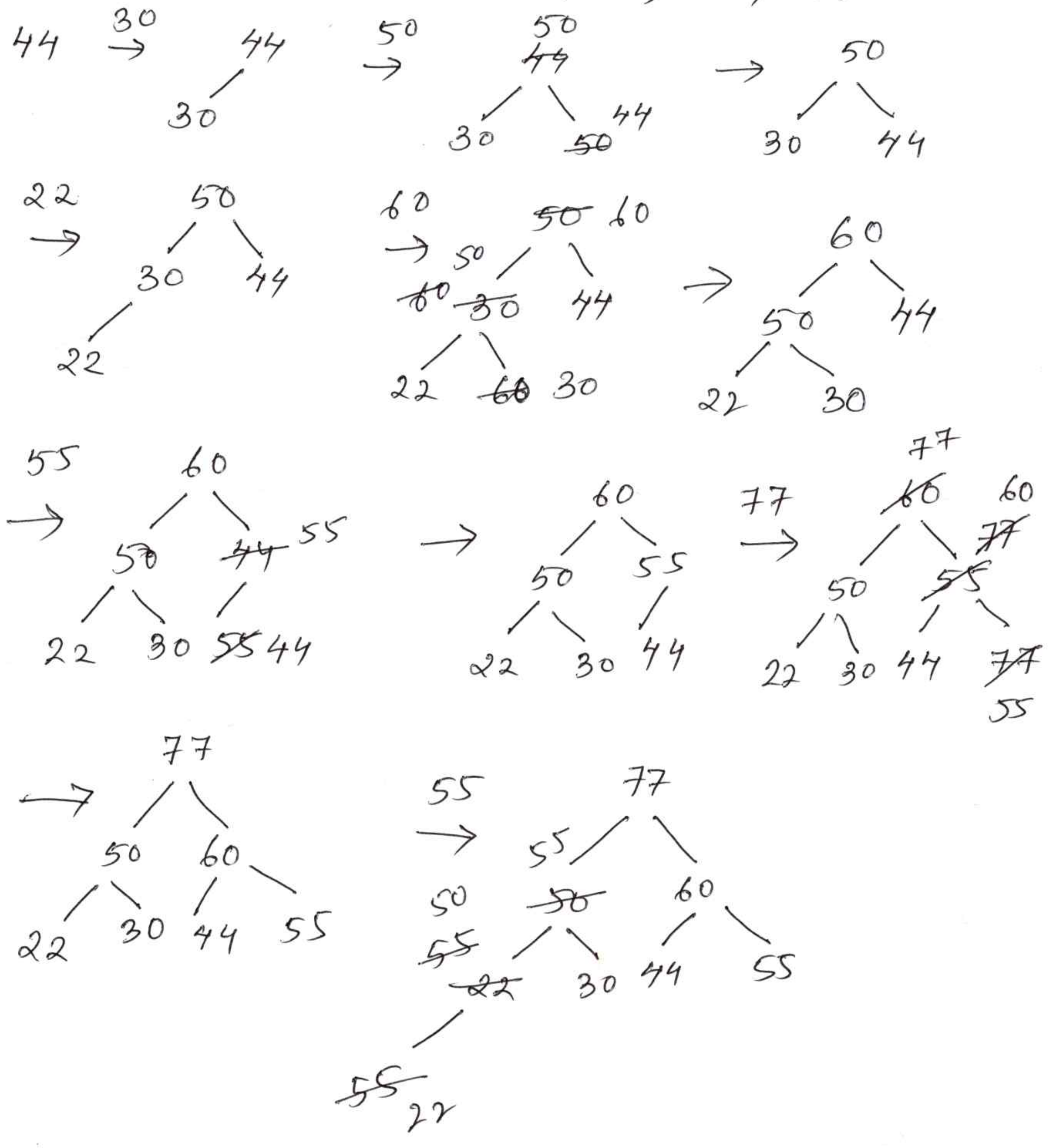
* For a binary tree with height h, $2^h - 1$ array space are required to store it.

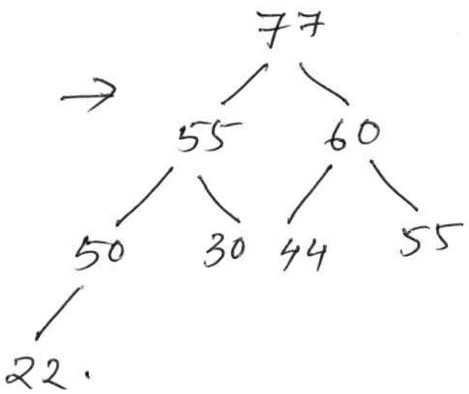
* Inserting into a heap

Suppose H is a heap with N elements, then an ITEM is inserted into the heap H as follows:

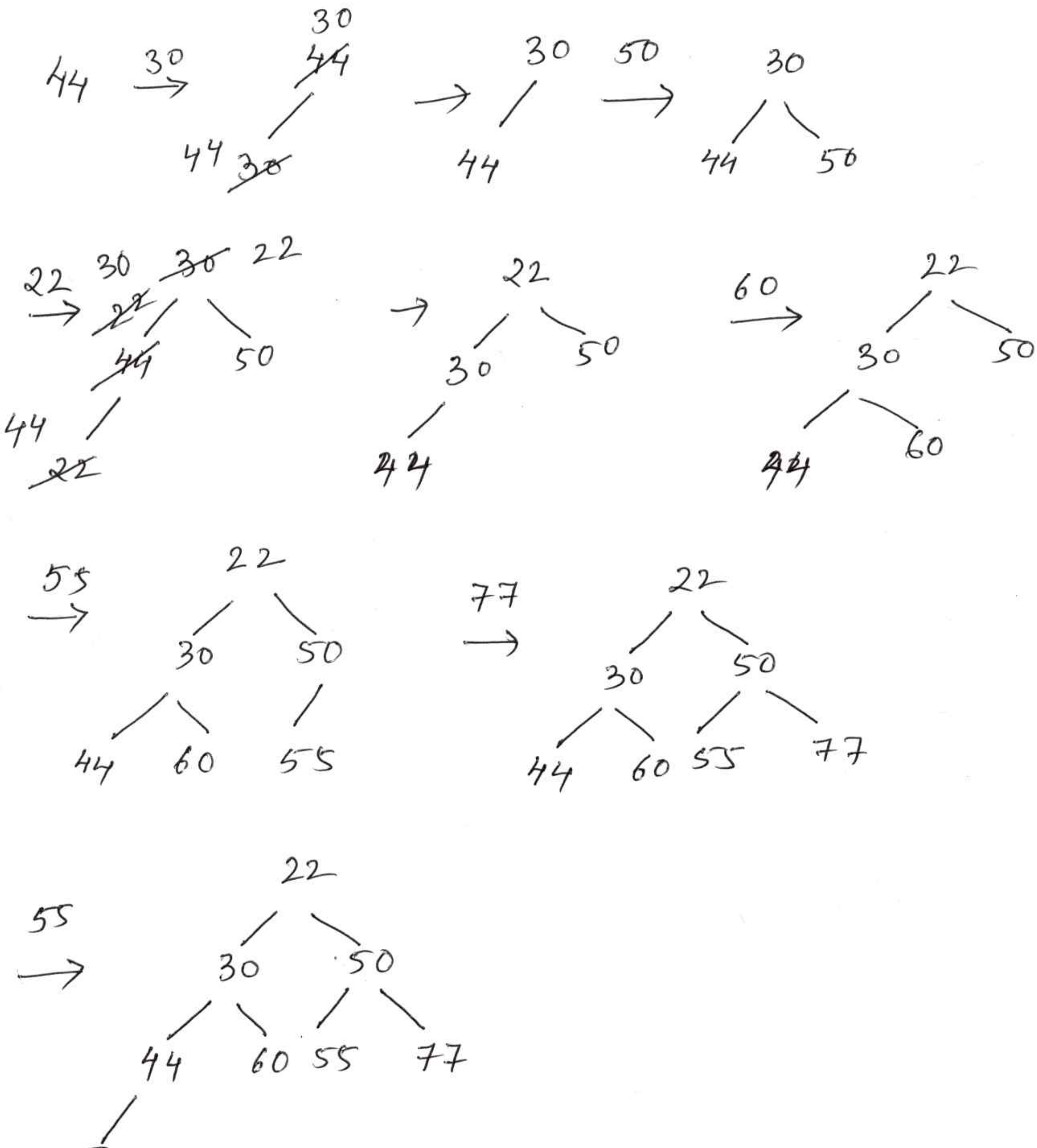
1. First adjoin ITEM at the end of H so that H is still a complete tree, but not necessarily a heap.
2. Then let ITEM rise to its appropriate place in H so that H is finally a heap.

(*) Construct a heap/maxheap with the following elements: 44, 30, 50, 22, 60, 55, 77, 55.





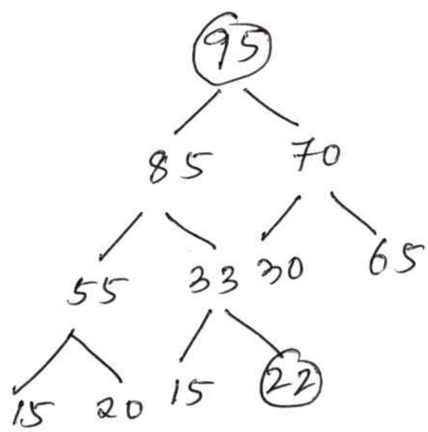
(*) Construct a min heap with the following elements : 44 , 30 , 50 , 22 , 60 , 55 , 77 , 55



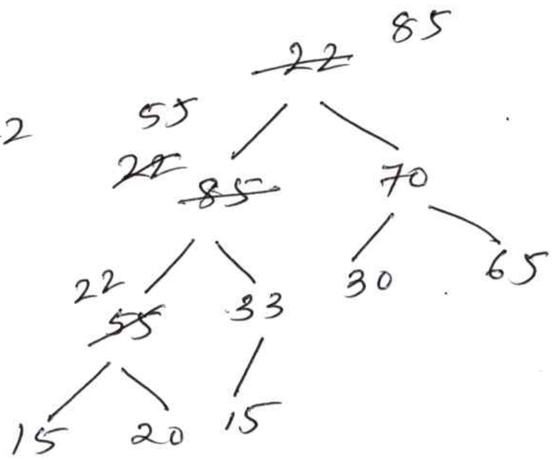
(*) Deleting the root of a heap

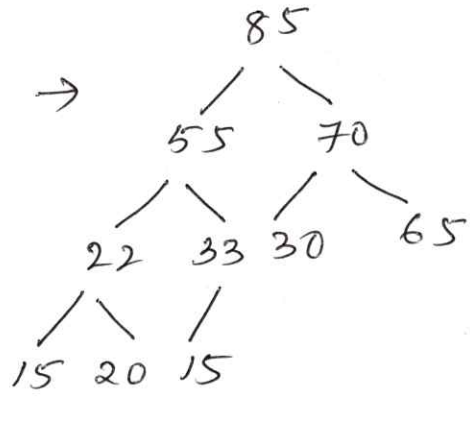
Suppose H is a heap with N elements, then the root R is deleted from H as follows:

1. Replace the deleted node R by the last node L of H so that H is still a complete tree, but not necessarily a heap.
2. (Reheap) Let L sink to its appropriate place in H so that H is still a heap.



delete 95
replace 95 with 22
→





⊛ Applications of Heaptree

1. Heapsort Implementation
2. Priority queue Implementation.

⊛ Sorting using a heaptree:

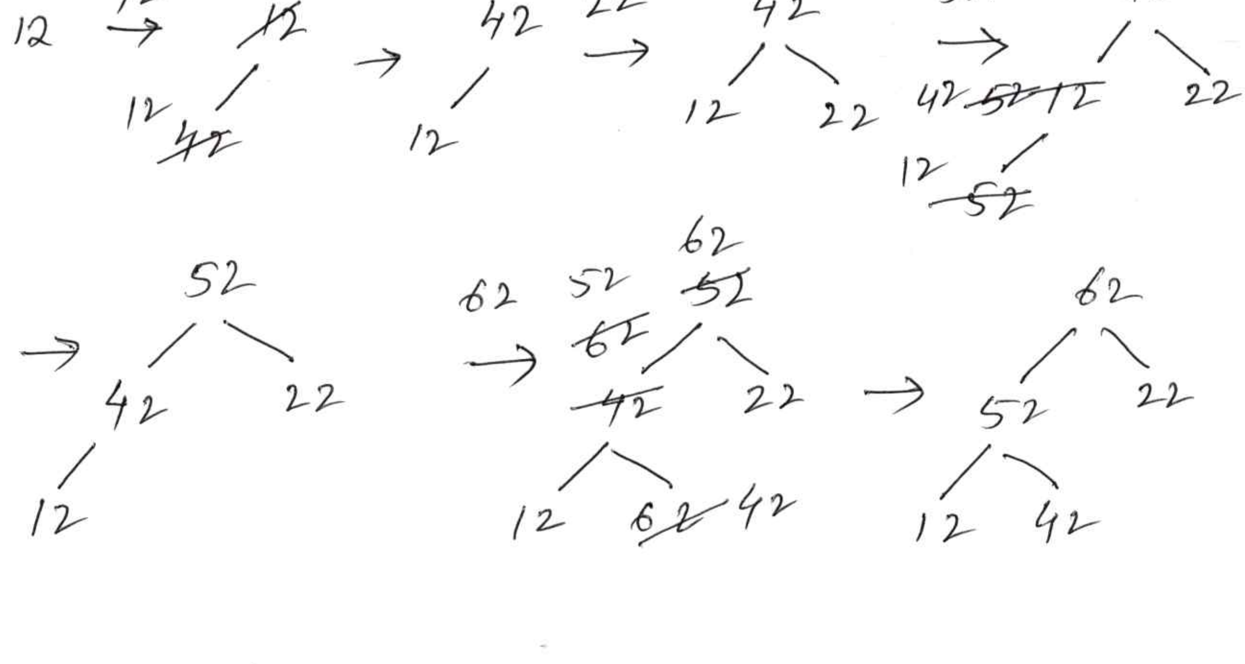
The heapsort algorithm to sort A (an array with N elements) consists of the following two phases:

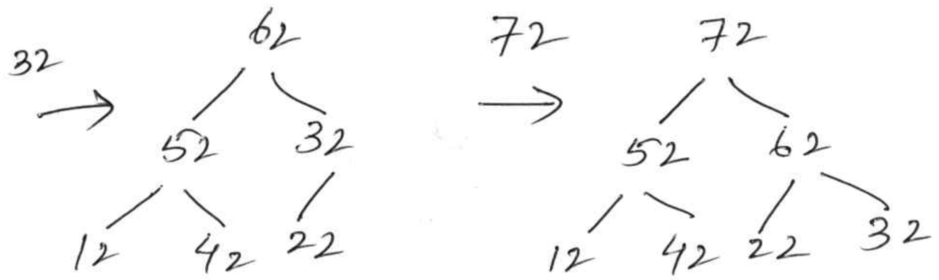
Phase I: Build a maxheap H out of the elements of A.

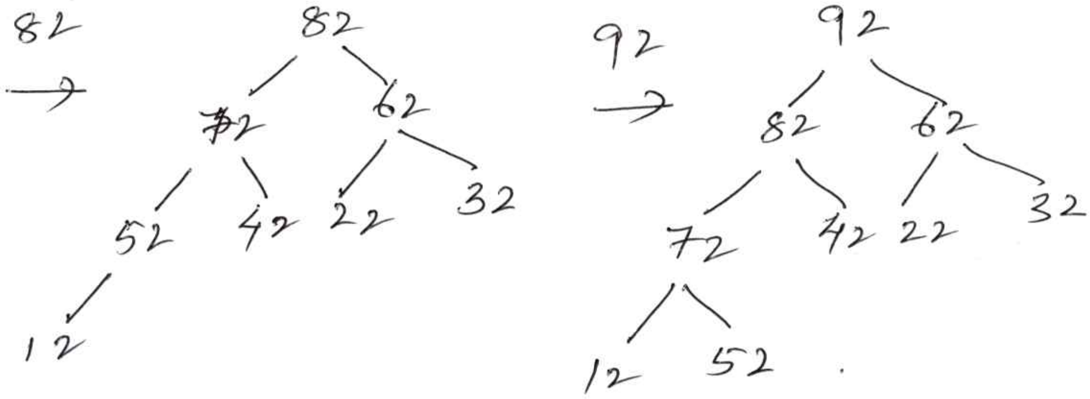
Phase II: Repeatedly delete the root element of H until the heap tree is empty.

⊛ Sort the following set of data in ascending order: 12, 42, 22, 52, 62, 32, 72, 82, 92.

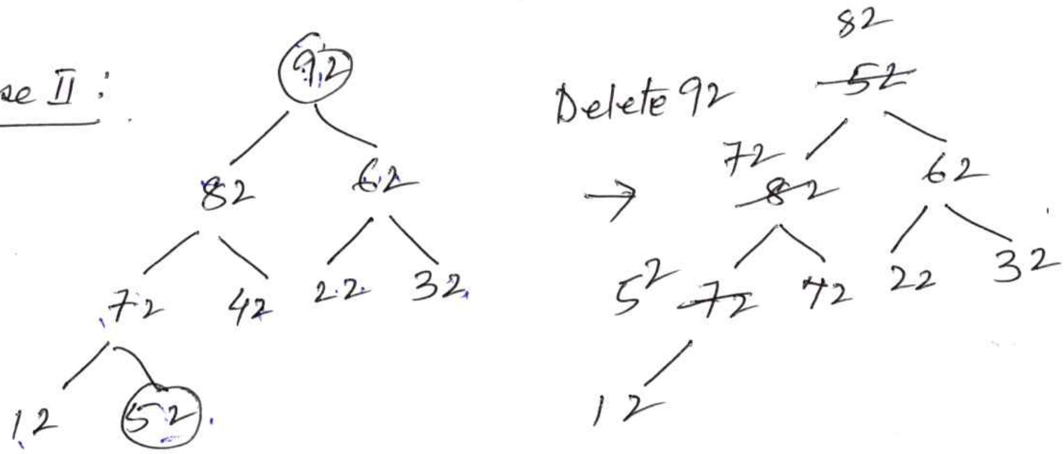
Phase I:







<u>Phase II</u>:



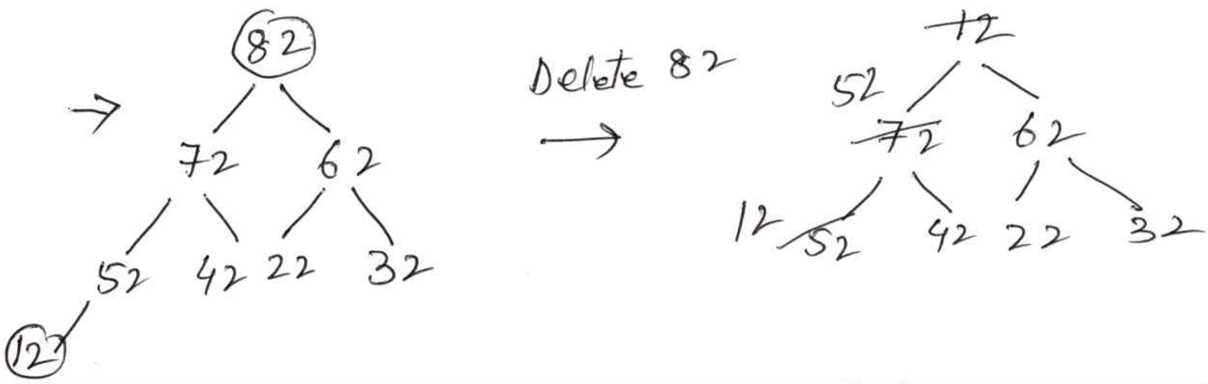
92	82	62	72	42	22	32	12	52

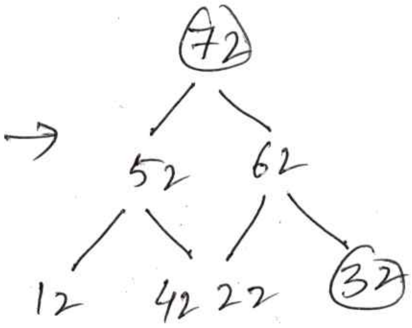
→ Swap 52 & 92

52	82	62	72	42	22	32	12	92

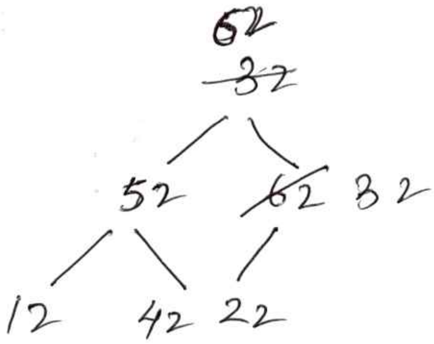
→ Reheap.

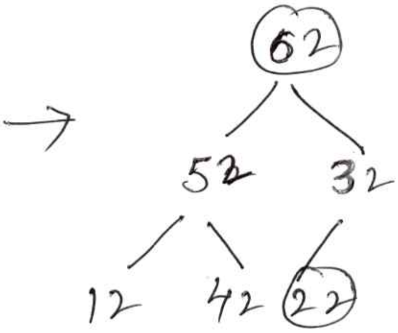
82	72	62	52	42	22	32	12	92



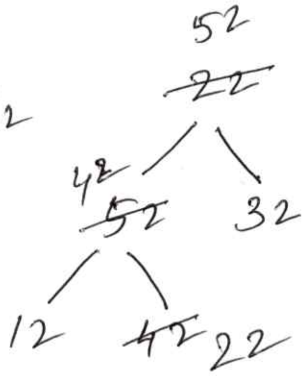


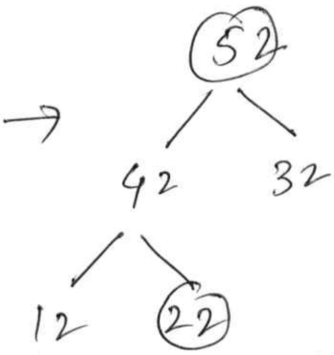
Delete 72



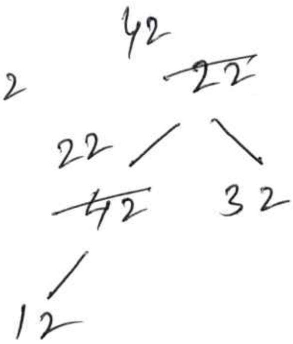


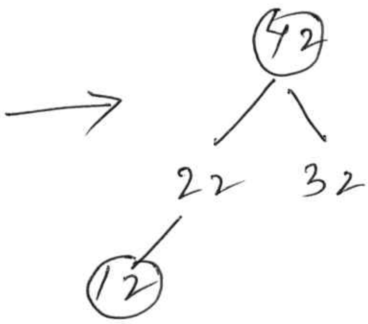
Delete 62



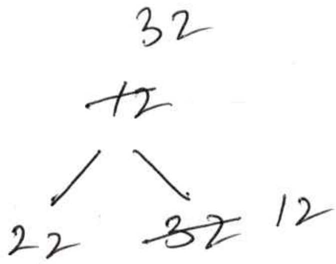


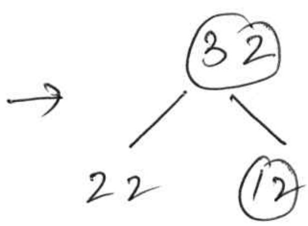
Delete 52



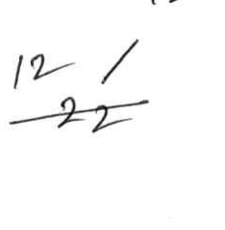


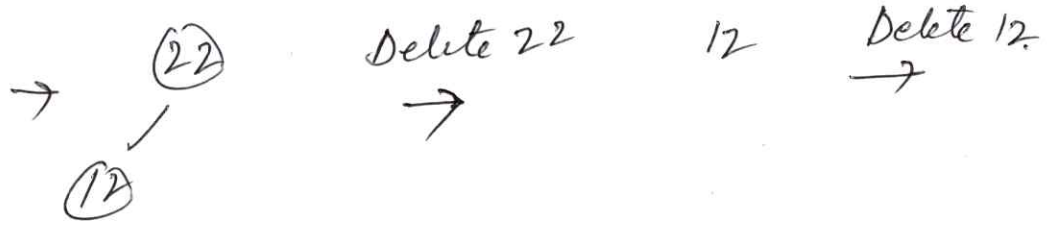
Delete 42





Delete 32





Sorted Array:

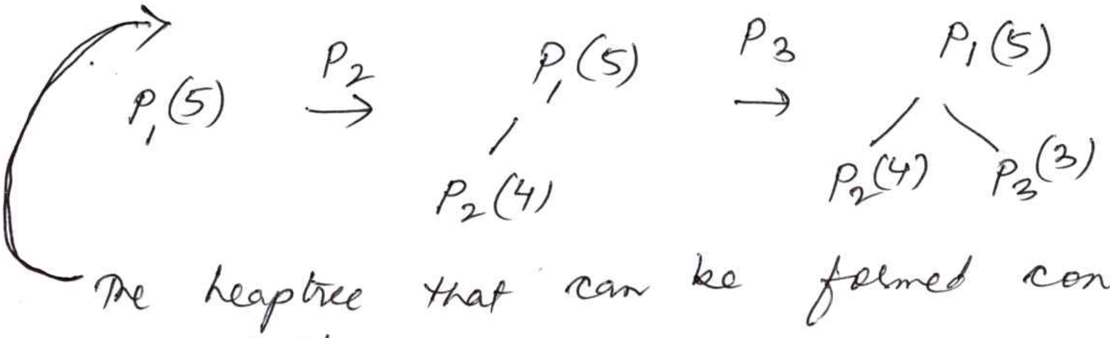
12	22	32	42	52	62	72	82	92

(*) Priority Queue Implementation using Heaptree.

Elements associated with their priority values are to be stored in the form of a heap tree, which can be formed based on their priority values. The top priority element that has to be processed first is at the root; so it can be deleted and the heap can be rebuild to get the next element to be processed and so on.

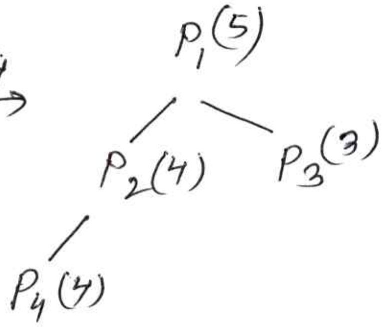
As an illustration, consider the following processes with their priorities:

Process:	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}.
Priority:	5	4	3	4	5	5	3	2	1	5.

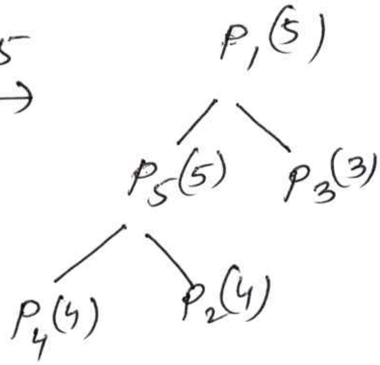


The heaptree that can be formed considering the priority values of the processes is shown below

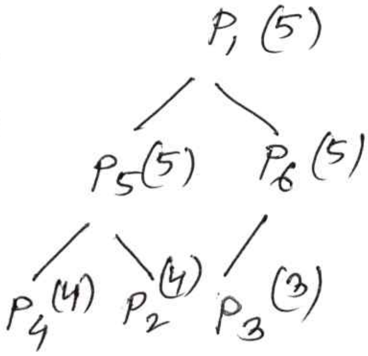
$P_4 \longrightarrow$



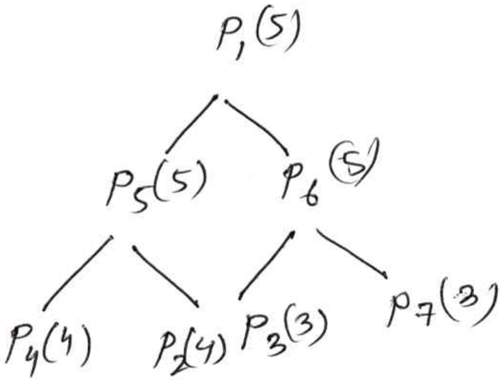
$P_5 \longrightarrow$



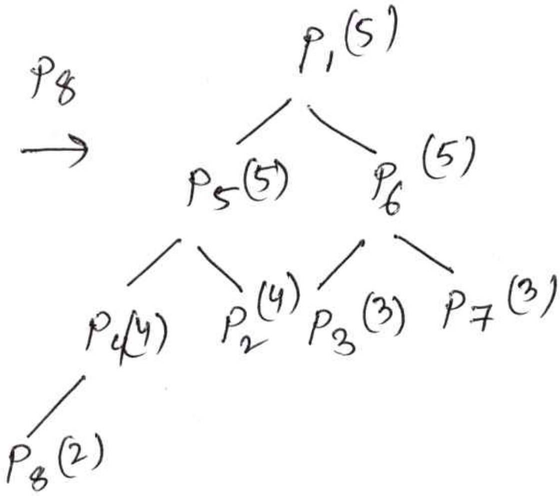
$P_6 \longrightarrow$



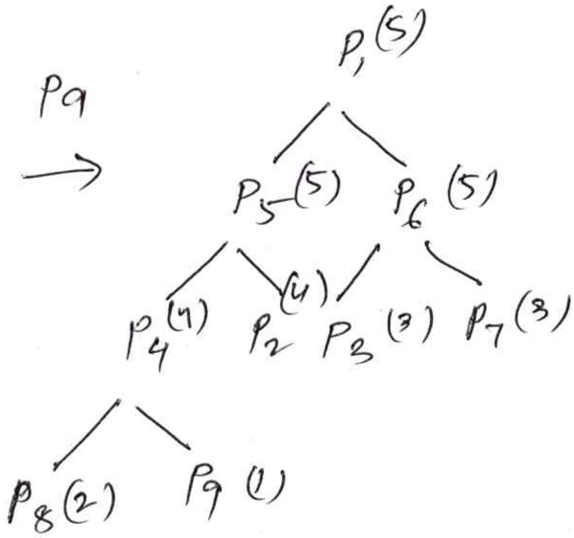
$P_7 \longrightarrow$



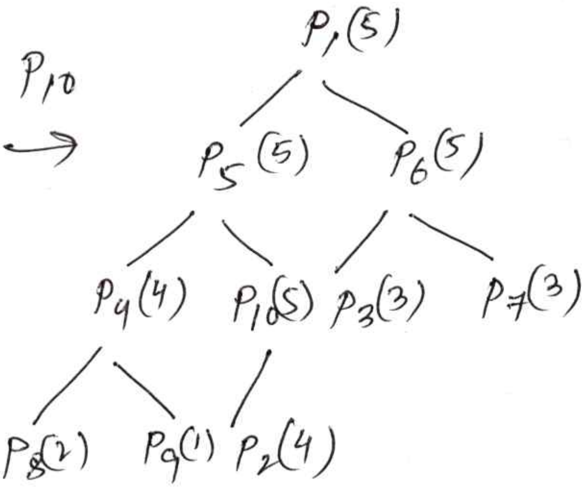
$P_8 \longrightarrow$



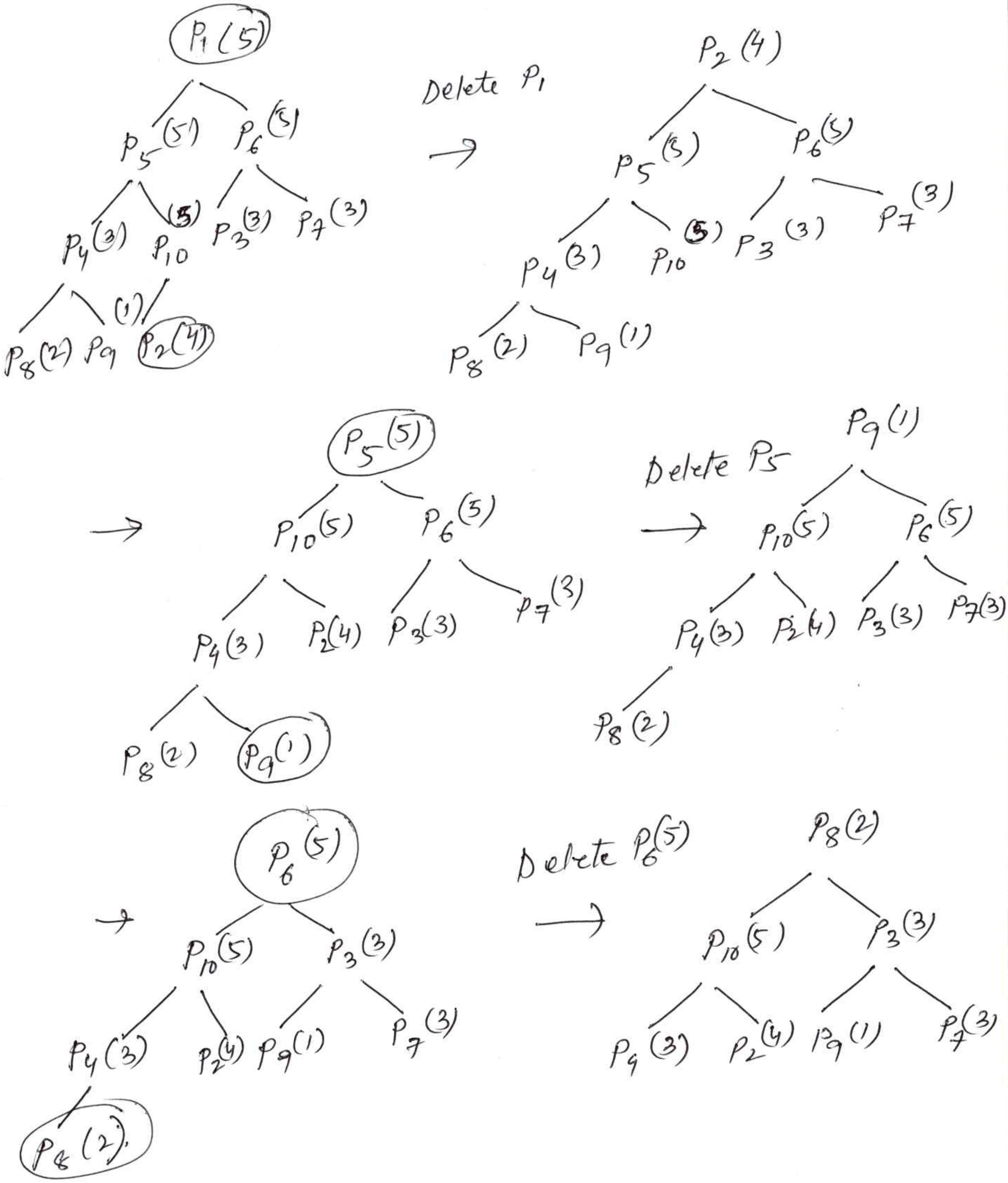
$P_9 \longrightarrow$

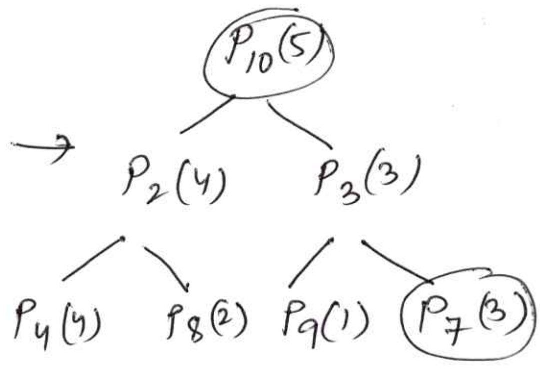


$P_{10} \longrightarrow$

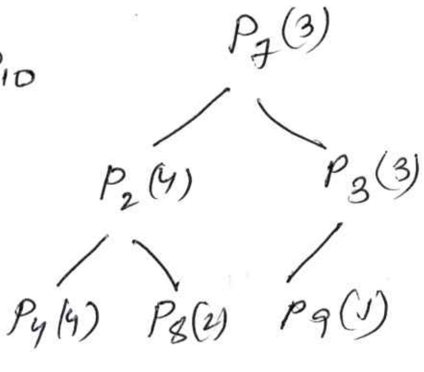


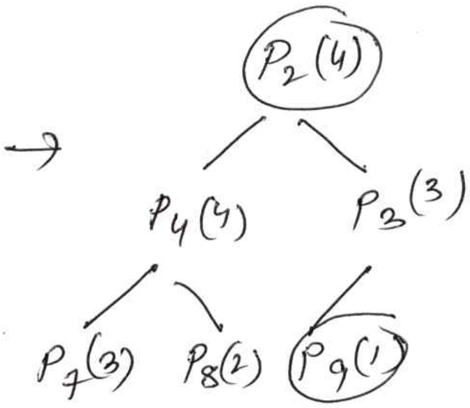
The order of servicing the processes is the ~~on~~ successive deletion of roots from the heap as illustrated below:



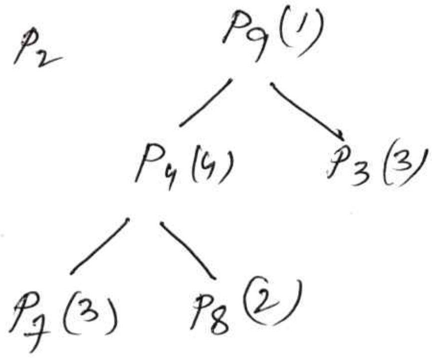


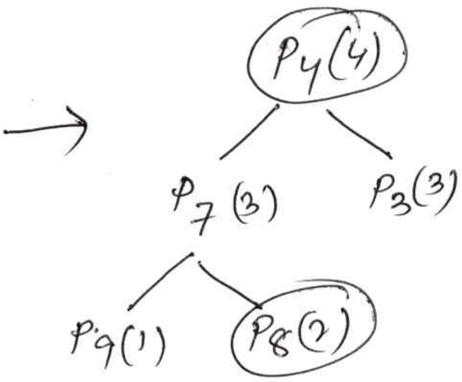
Delete P_{10} $\longrightarrow$



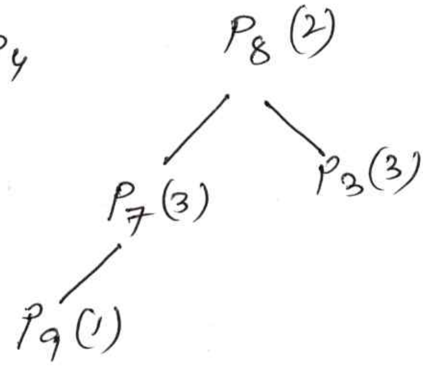


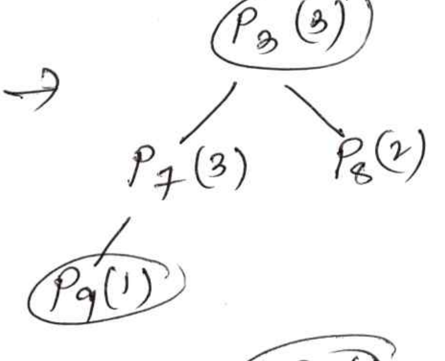
Delete P_2 $\longrightarrow$



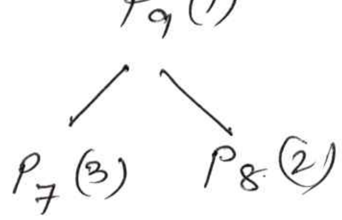


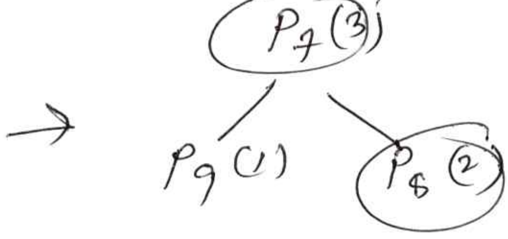
Delete P_4 $\longrightarrow$



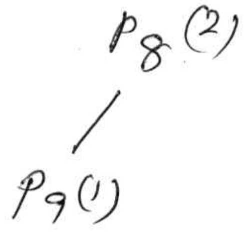


Delete P_3 $\longrightarrow$





Delete P_7 $\longrightarrow$



Delete P_8 $\longrightarrow$ $P_9(1)$

Delete P_9. $\longrightarrow$

(*) Complexity Analysis of Heapsort and Comparison with Mergesort and Quicksort.

Red Black Tree

A BST is called a RBT if it satisfies the following four properties:

1. Every node is either red or black.
2. The root and leaves are black.
3. If a node is red, then both its children are black (Red constraint)
4. Every simple path from a node to a descendent leaf contains the same number of black nodes (Black constraint)

(*) Black Height of a node

- Number of black nodes on any path from, but not including a node x to a leaf is called the black height of x and is denoted by $bh(x)$.

- Black height of a RBT is the black height of its root.

- If a child of a node does not exist, the corresponding pointer field will be NULL and these are regarded as leaves.

- Each node of a RBT contains the following fields:

colour	key	left	right	parent

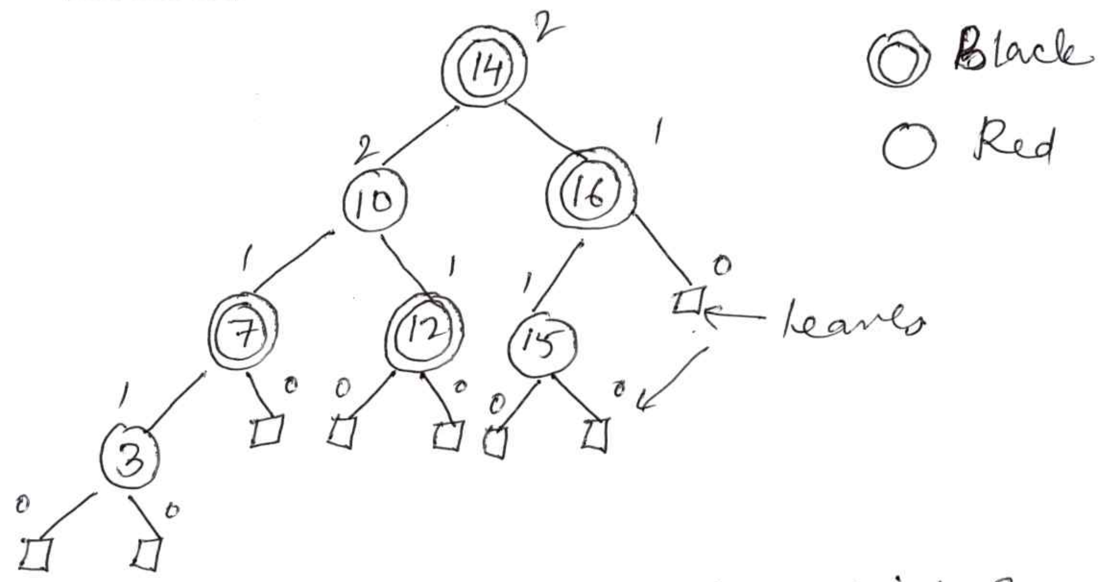


Figure : RBT of Black height 2.

Since RBT is a BST and operations that dont change the structure of a tree won't affect whether the tree satisfies the RBT properties such as search and traverse.

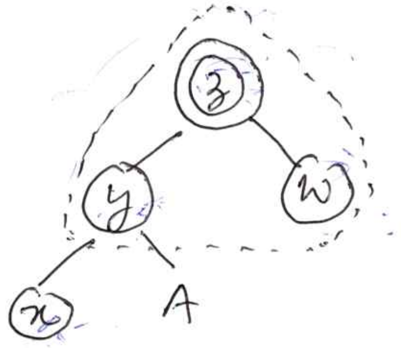
⊛ Inserting a key in a RBT

1. Use the BST insert algorithm to add a new node to the tree.
2. Colour the new node red.
3. Restore the RBT properties (if necessary)

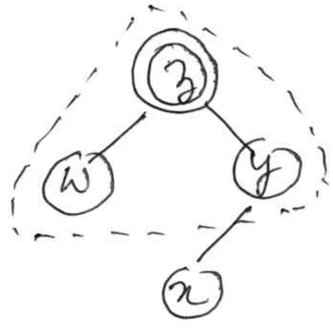
We will consider three cases. In each

diagram, the objects shown are subtrees of the entire RBT. There may be nodes above it or nodes below it, except where otherwise specified. A and B are subtrees and w, x, y, and z are single nodes.

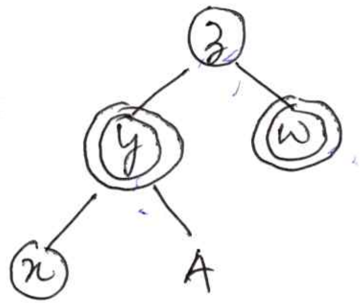
- <u>Case 1:</u> x's uncle is red.

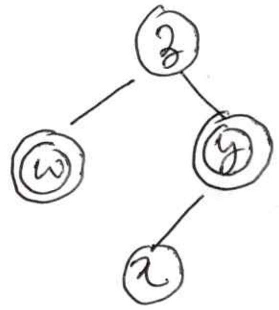


OR



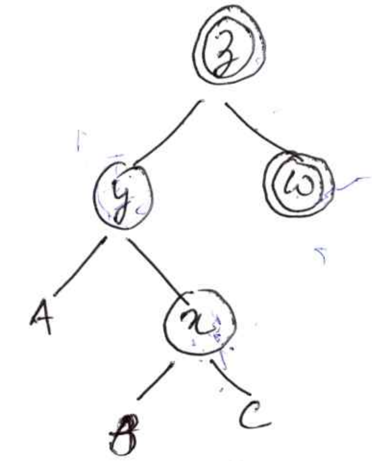
⇓ Recolour.



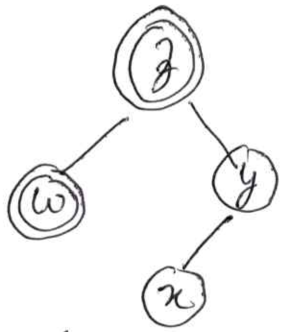


The problem here is that z's parent might be red, so we still have a violation of Red constraint. But we have moved the conflict ~~upo~~ upwards in the tree. Either we can keep applying case 1 until we reach the root, or we can apply case 2 or case 3 and end the fix up process. If we reach the root by applying case 1 (in other words, parent(z) is the root and parent(z) is red, then we just change parent(z) to black.

- Case 2: x's uncle is not red and $key(y) \le key(x) \le key(z)$ (or $key(y) \geq key(x) \geq key(z)$)

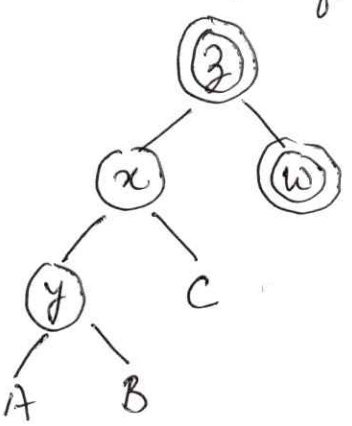


OR

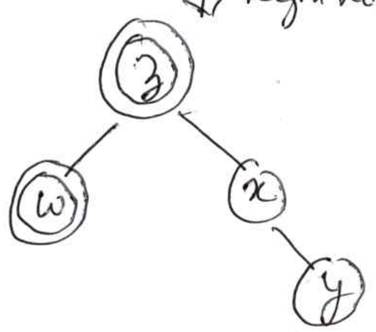


$\Downarrow$ Rotate around the parent

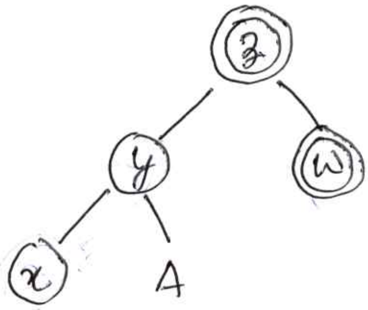
$\downarrow$ LeftRotate(y)



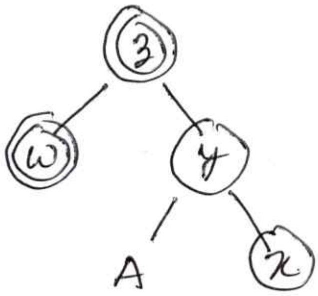
$\downarrow$ RightRotate(y)



- Case 3: x's uncle is not red (its black or does not exist) and $key(x) \le key(y) \le key(z)$ (or $key(x) \geq key(y) \geq key(z)$).

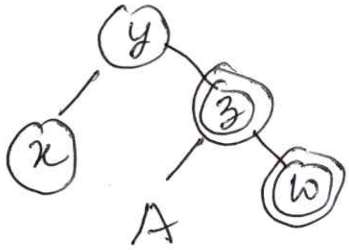


OR

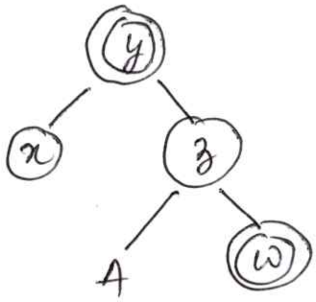


$\Downarrow$ Rotate around grandparent and recolour.

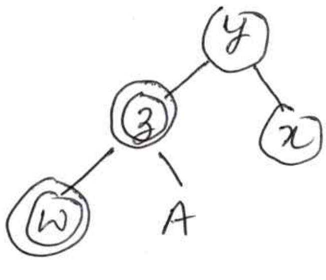
↓ Right Rotate (z)



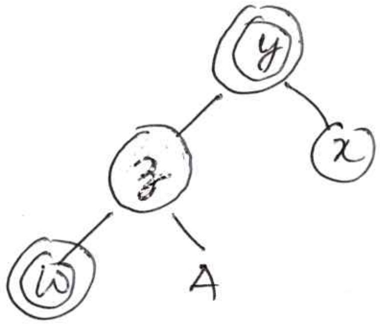
↓ Swap colour of parent & grandparent.



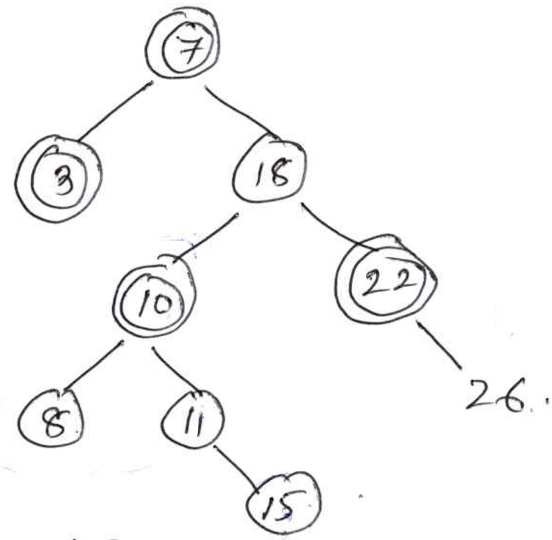
↓ Left Rotate (z)



↓ Swap colour of parent & grandparent

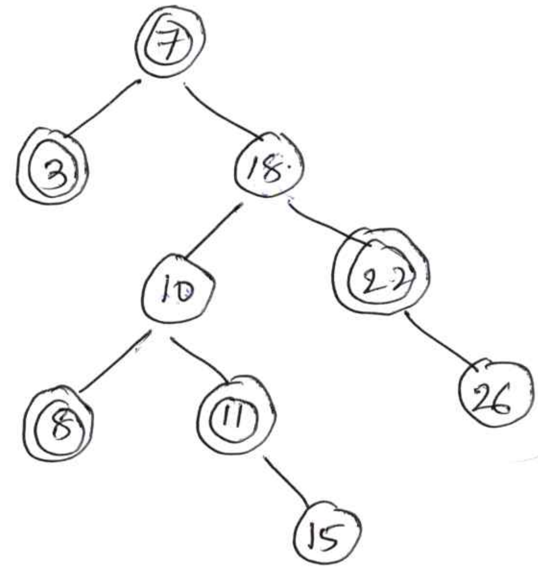


⊛ Example:

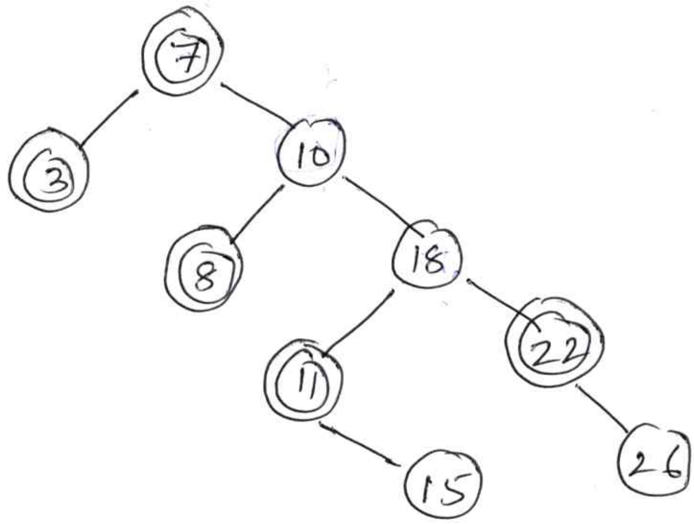


Insert $x = 15$

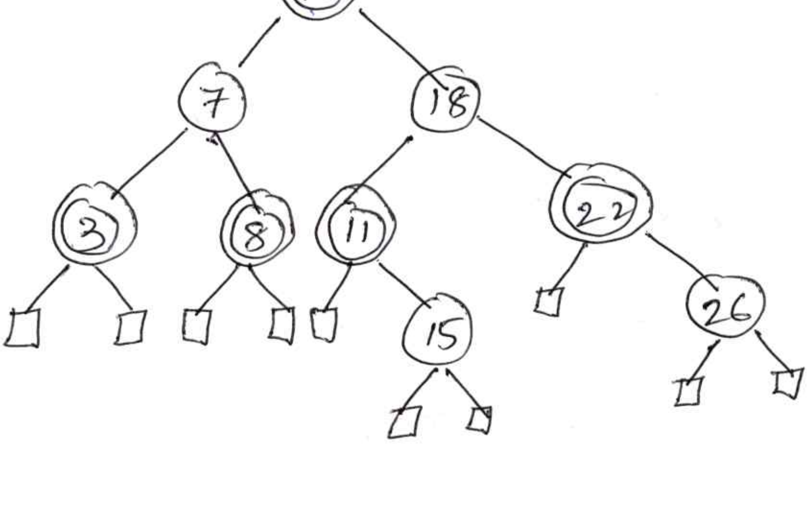
<u>Case 1</u>: x's uncle is red, recolour parent, uncle and grandparent nodes, moving the violation up the tree.



Case 2: Right Rotate (18)



Case 3: Left Rotate (7) & Recolour.



(7) Construct a Red Black tree with the following elements : 1, 2, 3, 4, 5, 6.