Fixed - Length codes.

Suppose we want to compress a 100,000 byte data file that we know contains only the uppercase letters A through F. Since we have only six distinct characters to encode, we can represent each one with three bits rather than the eight bits normally used to store characters.

Letter	A	B	C	D	E	F
Code_word	000	001	010	011	100	101

Compression Ratio = 5/8 = 62.5 %

Variable length code

What if we knew the relative frequencies at which each letter occurred? It would be logical to assign shorter codes to the most frequent letters and save longer codes for the infrequent letters.

eg

Letter	A	B	C	D	E	F
Frequency	45	13	12	16	9	5
Code word	0	101	100	111	1101	1100

Using this code, our file can be represented with

$(45\times1+13\times3+12\times3+16\times3+9\times4+5\times4)\times1000$

$= 224000$ bits (or) 28000 bytes.

Compression Ratio = 72%.

Huffman Coding

Huffman Coding is a lossless data compression algorithm. The idea is to assign variable length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the ~~big~~ largest code.

The variable-length codes assigned to input characters are Prefix codes (bit sequence) codes are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman coding makes sure that there is no ambiguity when decoding the generated bit stream.

Example

Consider four characters a, b, c and d, and their corresponding variable length code be 00, 01, 0 and 1.

a - 00

b - 01

c - 0

d - 1

This coding leads to ambiguity because code assigned to c is prefix of codes assigned to a and b. If the compressed bit stream is 0001, the decompressed output may be "cccd" or "ccb" or "acd" or "ab".

There are mainly two major parts in Huffman coding

1) Build a Huffman tree from input character
2) Traverse the Huffman tree and assign codes to characters.

Steps to build Huffman Tree

Input is array of unique characters along with their frequency of occurance and output is Huffman tree.

A Huffman tree is a special type of binary tree used for data Compression.

Lets say you have a set of characters and their frequency of use and want to create a huffman encoding for them.

eg

Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

The Simplest Construction algorithm uses a priority queue where the node with Lowest probability is given highest priority.

Step1: Create a leaf node for each symbol and add it to the priority Queue.

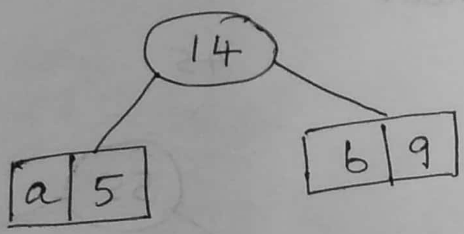
Step2: While there is more than one node in the queue:

i) Remove the two nodes of highest priority (Lowest probability) from the queue

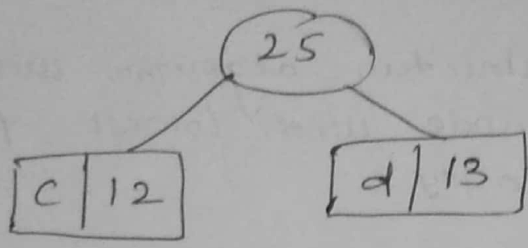
ii) Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes probabilities

iii) Add the new node to the queue

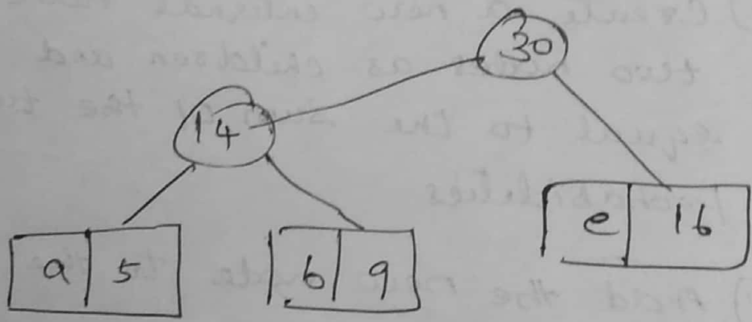
Step 3: The remaining node is the root node and the tree is complete.



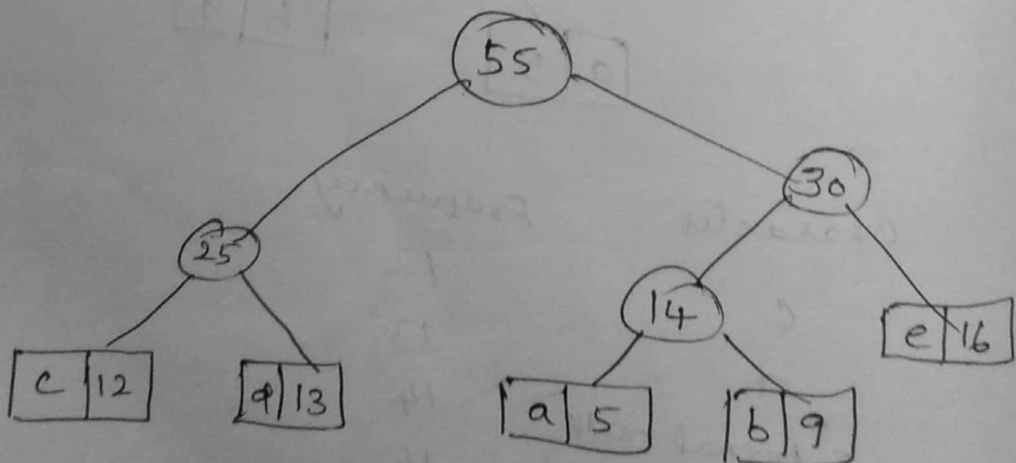
Character	Frequency
c	12
d	13
Internal node	14
e	16
f	45



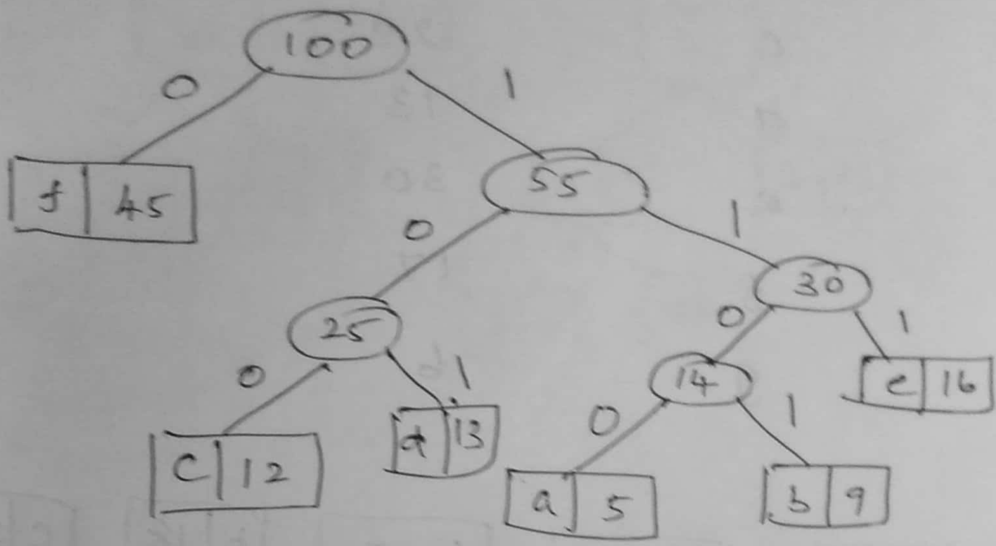
Character	Frequency
Internal node	14
e	16
Internal node	25
f	45



Character	Frequency
Internal node	25
Internal node	30
f	45



Character	Frequency
f	45
Internal node	55

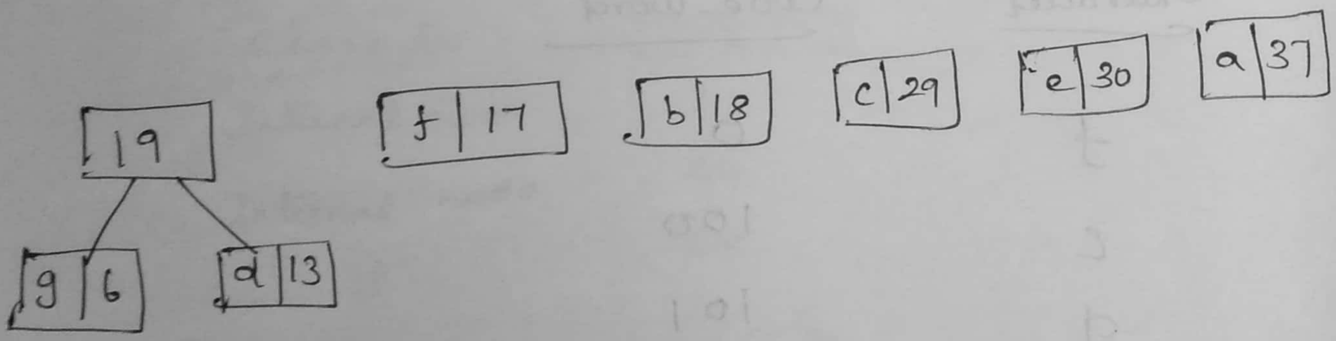


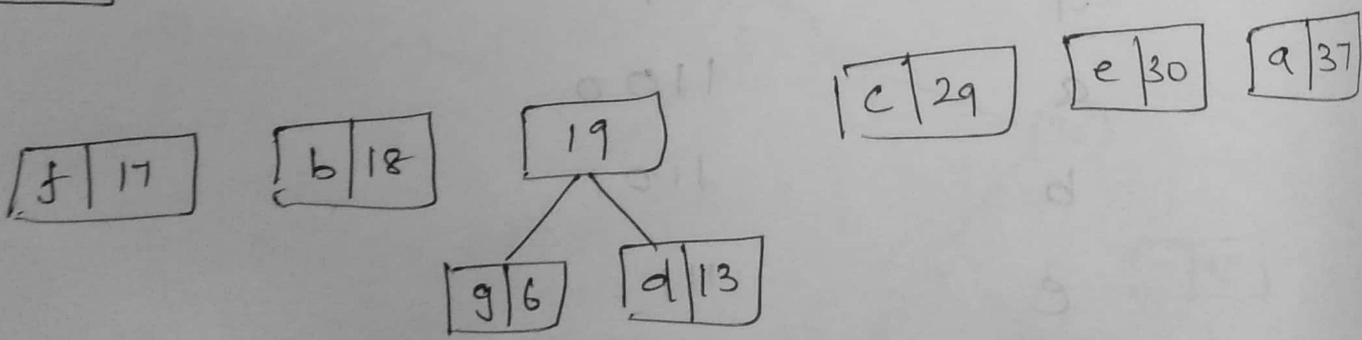
Character	Code-word
f	0
c	100
d	101
a	1100
b	1101
e	111

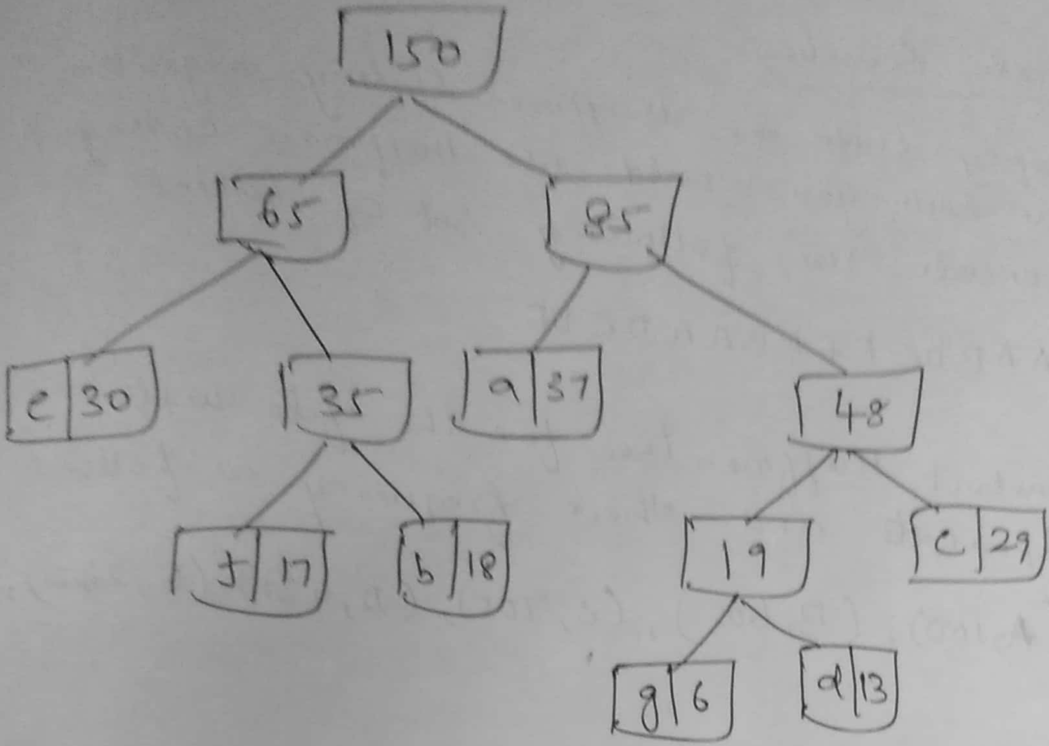
Example 2:

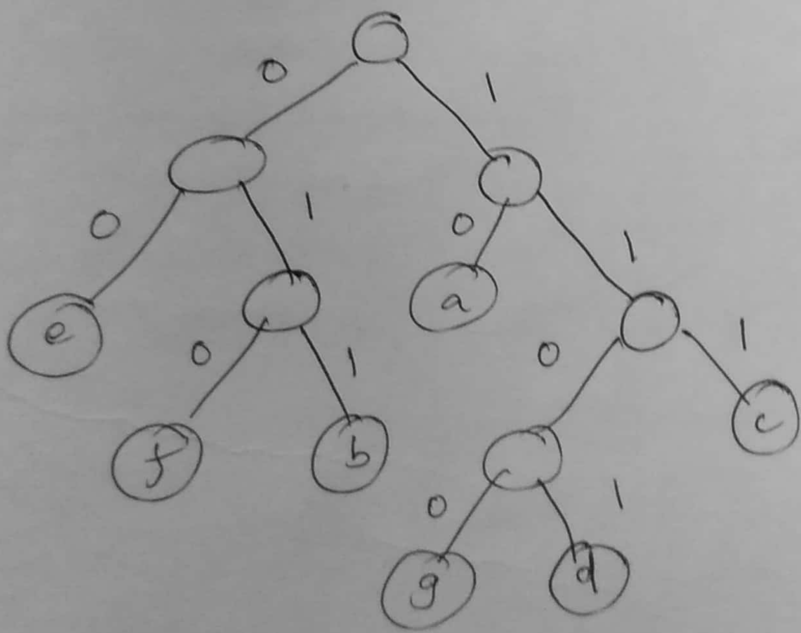
Character	Frequency
a	37
b	18
c	29
d	13
e	30
f	17
g	6

g, 6 | d, 13 | f, 17 | b \| 18 | c \| 29 | e \| 30 | a \| 37









a	10
b	011
c	111
d	1101
e	00
f	010
g	1100

Example Question

1. Briefly state the Huffman coding algorithm. Show how you would use Huffman coding to encode the following set of tokens

   AAABDC EFBBAADCDF

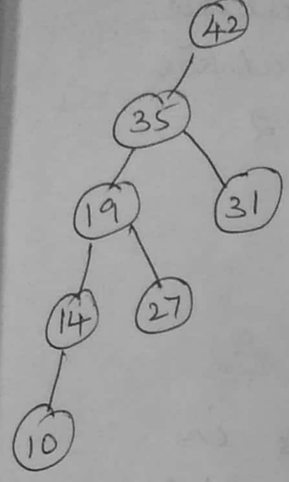
2. Construct Huffman tree for the file with alphabets and their frequency as follows

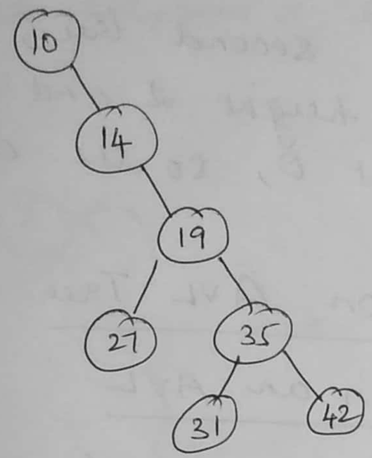
   {(A,100), (B,605), (C,705), (D,431), (E,242), (F,59)}

Skewed BST

The BST results in left or right skewed depending on the order of the list decreasing or increasing respectively.

example



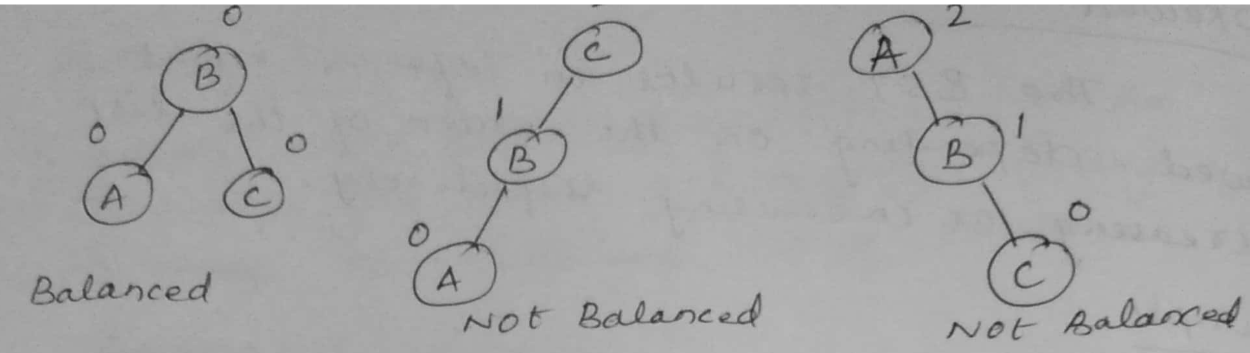


The disadvantage of skewed BST is that the worst case time complexity of a search is $O(n)$. Therefore, there arises the need to maintain BST to be of balanced height. By doing so, it is possible to obtain for the search operation a time complexity of $O(\log_2 n)$ in the worst case.

AVL Tree

Named after their inventor Adelson, Velski and Landis, AVL trees are height balancing binary search tree. AVL tree checks the height of the left and right subtrees and assures that the difference is not more than one(1). This difference is called the Balance Factor.

Balance Factor = height (left subtree) - height (Right subtree)



In the second tree, the left subtree of c has height 2 and the right subtree has height 0, so the difference is 2.

Operations on AVL Tree

Insertion in an AVL

Insertion is simple and same as in BST but it is done by keeping in mind the balanced factor. If balanced factor is affected after doing insertion, to render it we have to perform some rotations to restore the balanced factor of node. ~~In order to balance the tree, there are 4 cases of rotations such as~~

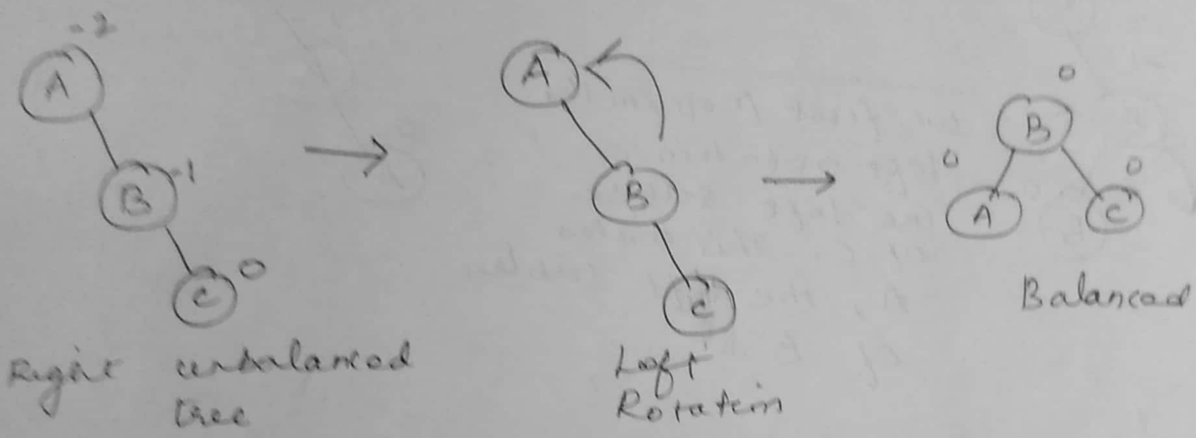
An AVL tree may perform the following four kinds of rotations.

- ↳ Left Rotation
- ↳ Right Rotation
- ↳ Left - Right Rotation
- ↳ Right - Left Rotation.

The first two rotations are single rotations and the next two rotations are double rotations.

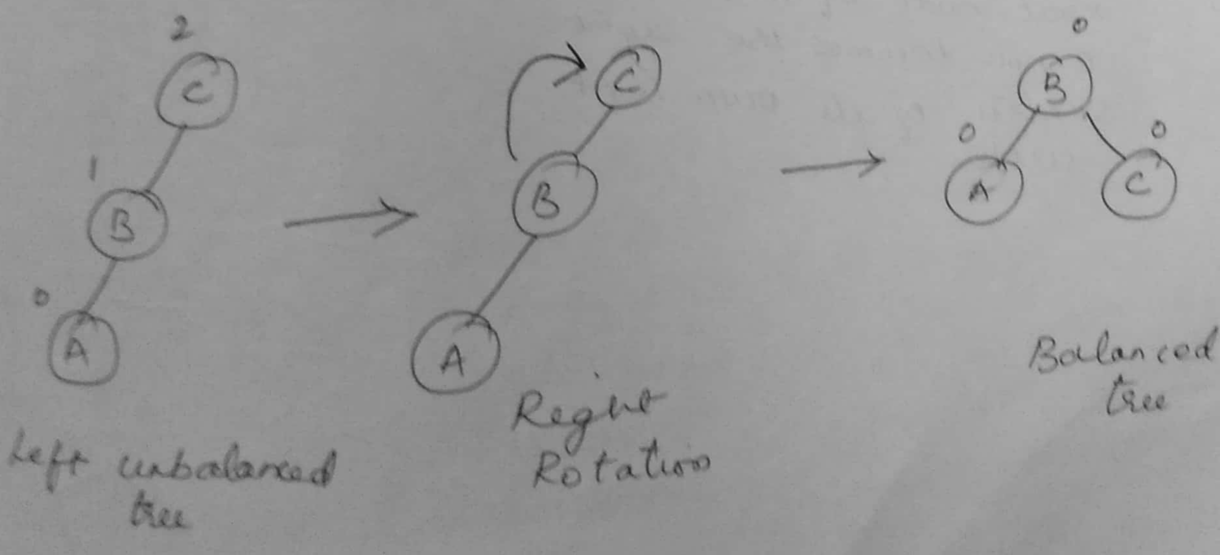
Left Rotation (LL)

If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation



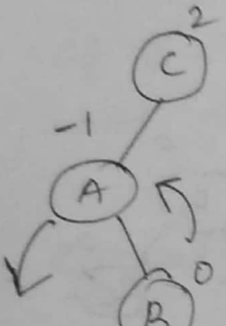
Right Rotation (RR)

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation

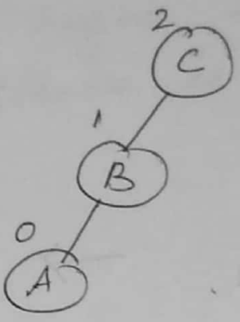


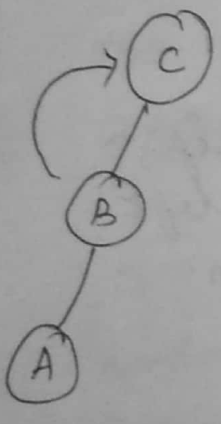
Left - Right Rotation (LR)

A node has been inserted into the right subtree of the left subtree. This makes C an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.

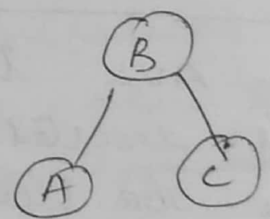


we first perform the left rotation on the left subtree of C. This makes A, the left subtree of B.



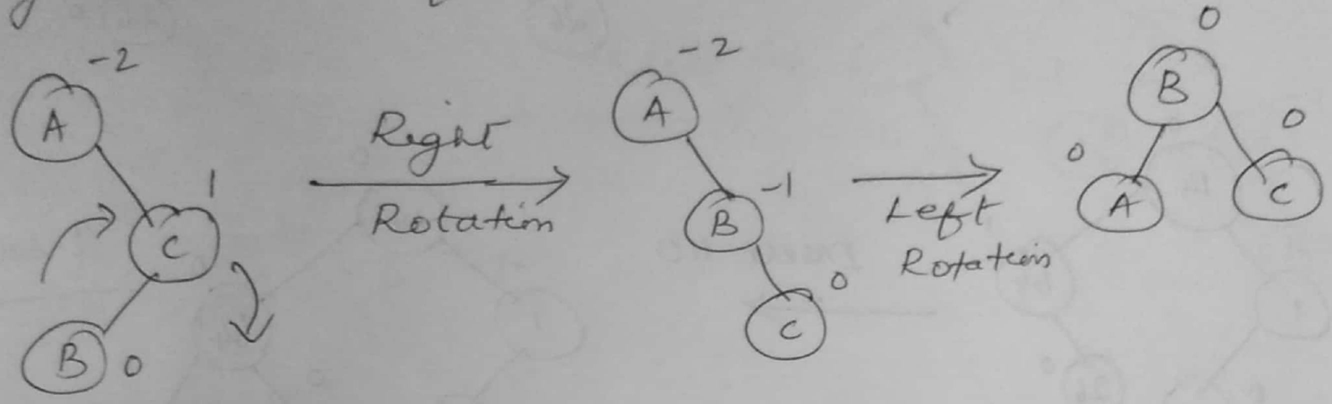


we shall now right-rotate the tree, making B the new root node of this subtree. C now becomes the right subtree of its own left subtree.



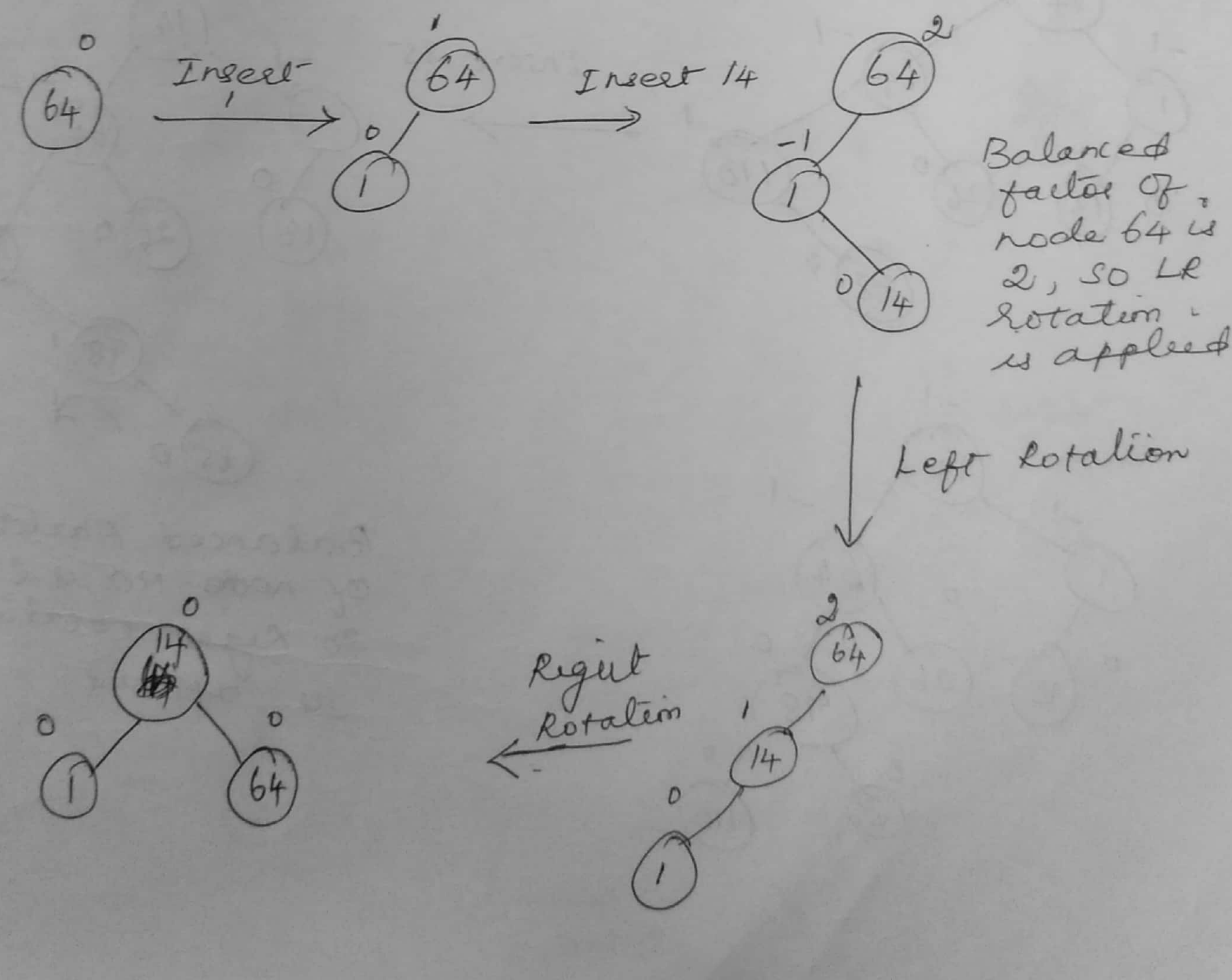
Right - Left Rotation (RL)

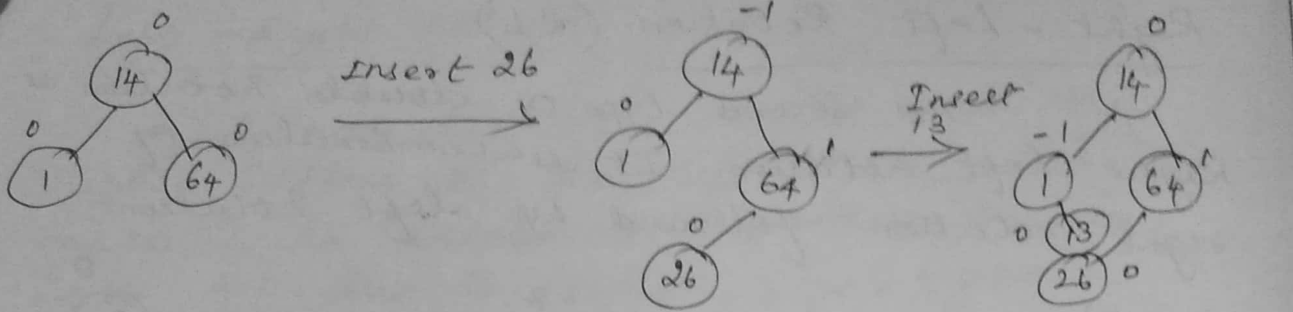
The second type of double rotation is Right-Left rotation. It is a combination of right rotation followed by left rotation.

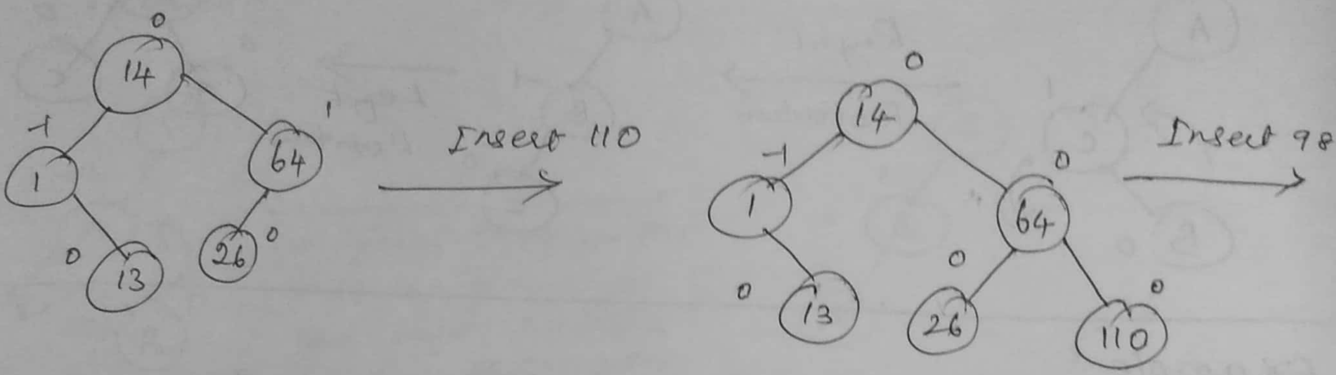


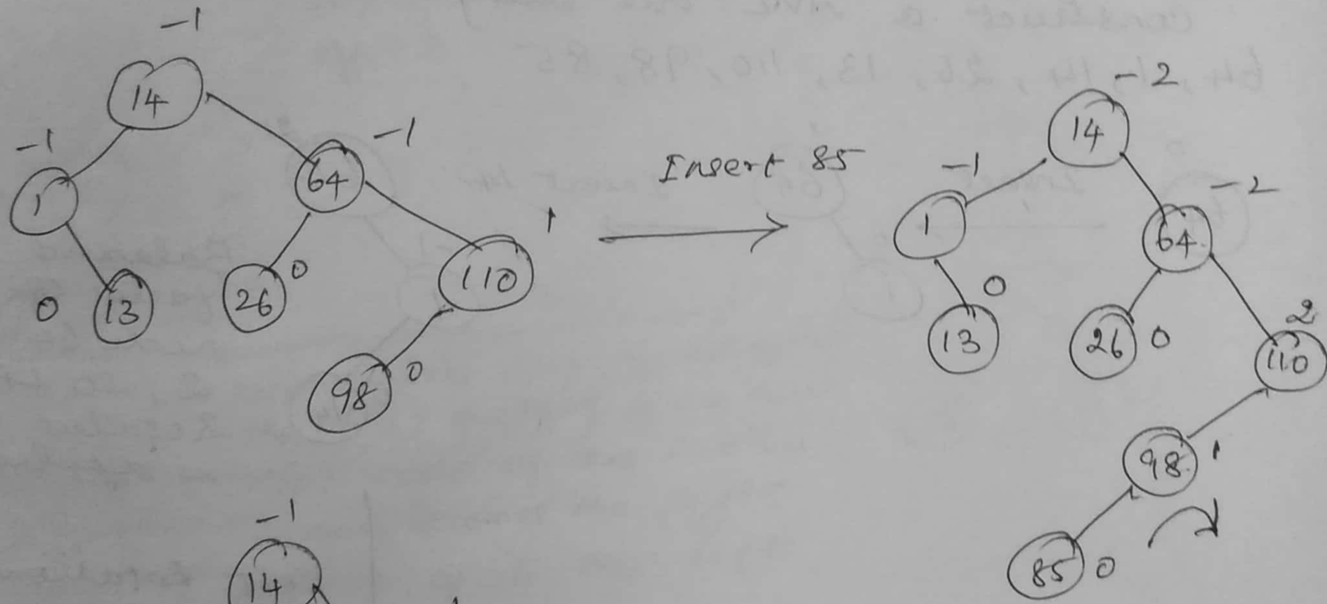
Example

Construct a AVL tree Using elements 64, 1, 14, 26, 13, 110, 98, 85.

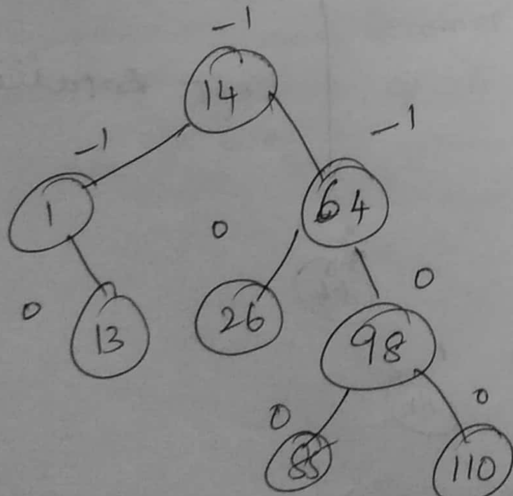






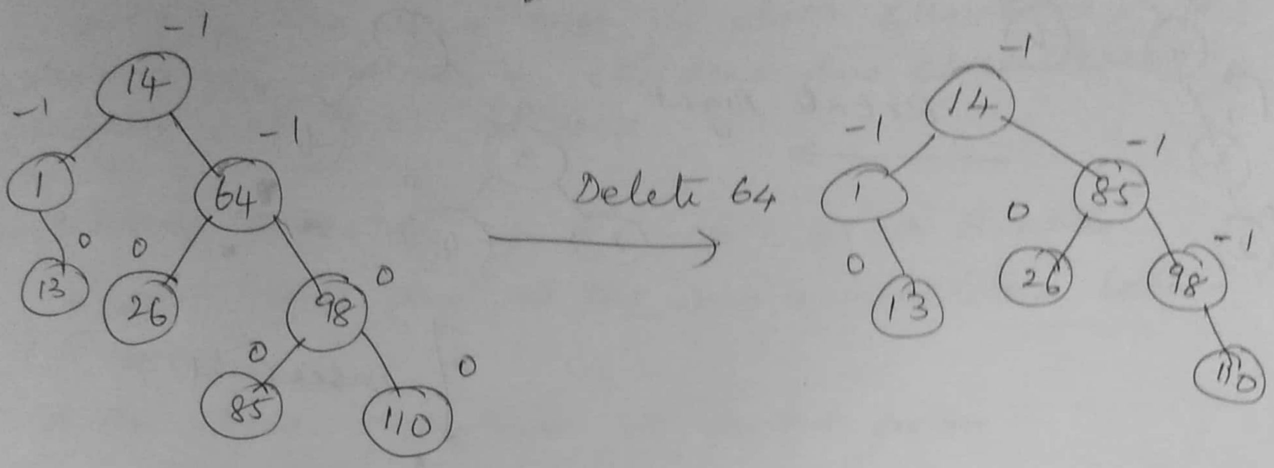


Balanced Factor of node 110 is 2 so Right rotation is applied



Delete

Now delete 64 from tree

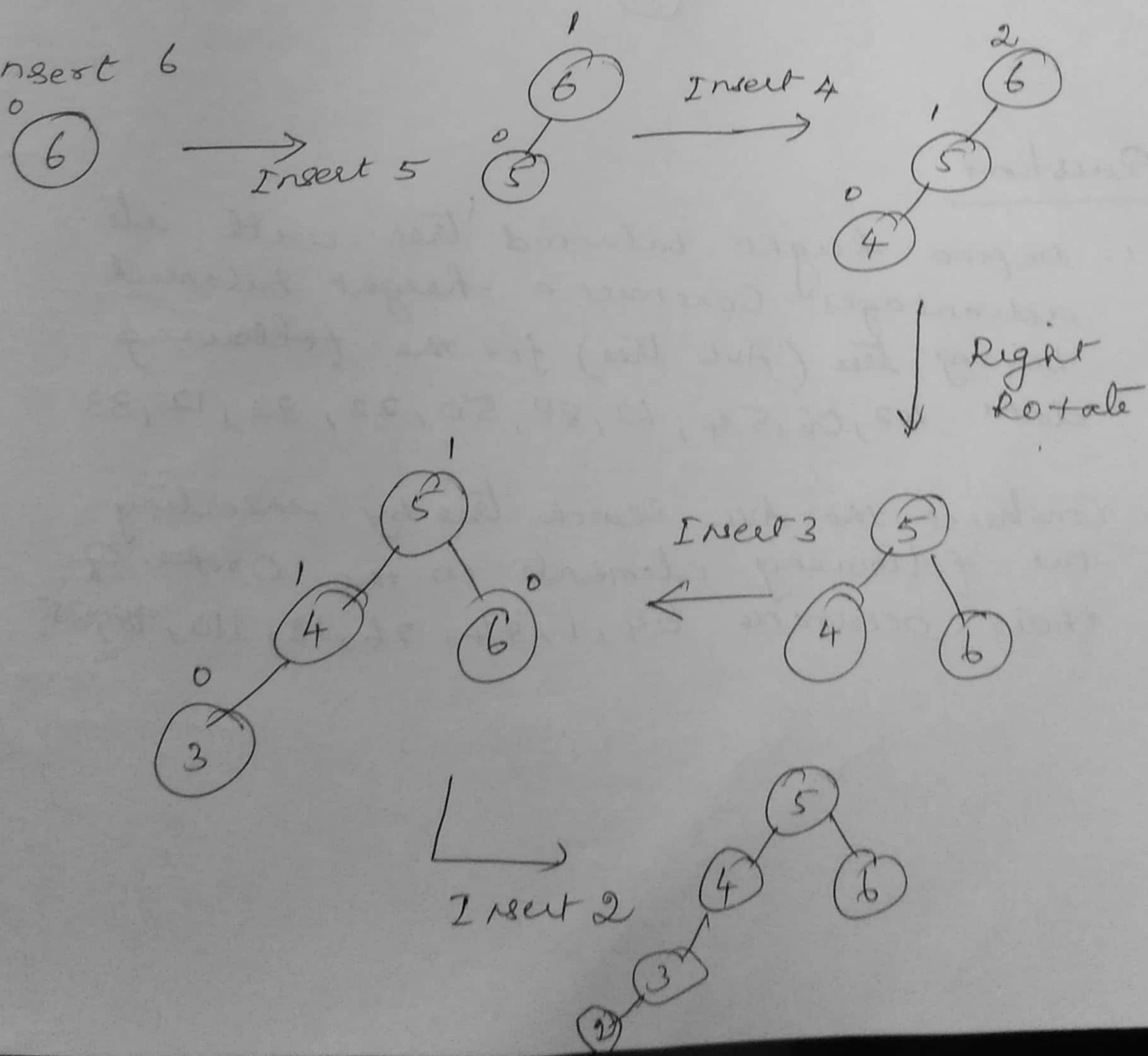


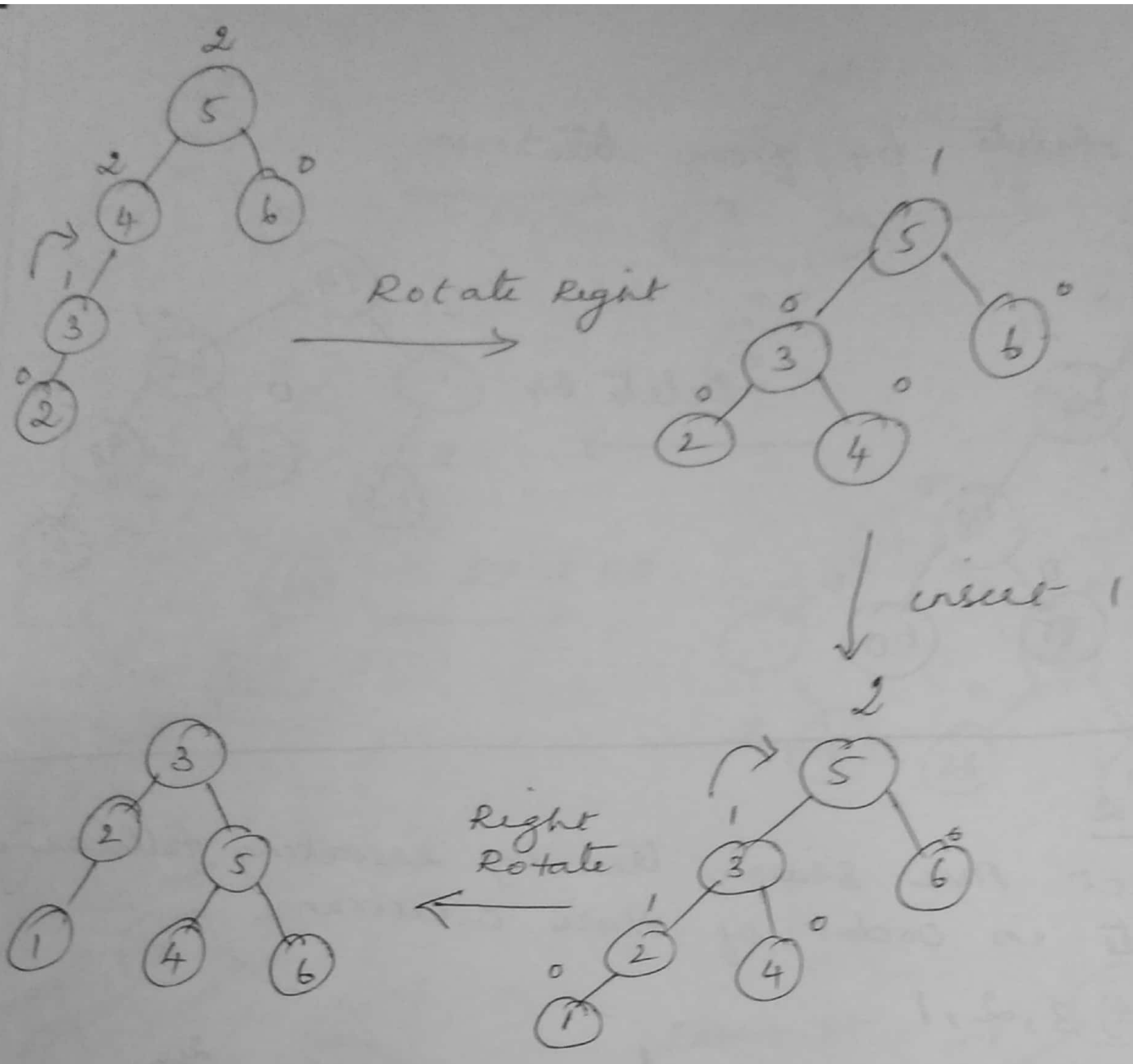
Example 2

Construct AVL search tree by inserting following elements in order of their occurrence

6, 5, 4, 3, 2, 1

Insert 6





Questions

1. Define height balanced tree with its advantages. Construct a height balanced binary tree (AVL tree) for the following data 42, 06, 54, 62, 88, 50, 22, 32, 12, 33

2. Construct the AVL search tree by inserting the following elements in the order of their occurance 64, 1, 44, 26, 13, 110, 98, 85

2-3 Tree

A 2-3 tree is a type of data structure, where every node with children has either two children or three children.

* Every non-leaf is a 2-node or a 3-node
* All leaves are at the same level (the bottom level)
* All data are kept in sorted order
* Every non-leaf node will contain 1 or 2 fields.

Splay Trees - Self-adjusting BST

Splay trees are binary search trees that have implemented a self-adjusting mechanism. This mechanism performs in the following way: every time we access a node of the tree, whether for insertion or retrieval, we perform rotations, lifting the newly inserted/accessed node all the way up, so that it becomes the root of the modified tree. The nodes on the way are rotated such that the tree becomes more balanced.

A splay operation on a binary search tree moves a designated node to the root of the tree by doing sequence of rotations along the path from the node to the root.

The rotations occur in pairs, mostly but not completely bottom up. We denote by $P(x)$ the parent of node 'x'. A rotation at 'x' replaces $P(x)$ by 'x' and makes the old $P(x)$ a child of x, the right (left) child if x was a left (right) child.

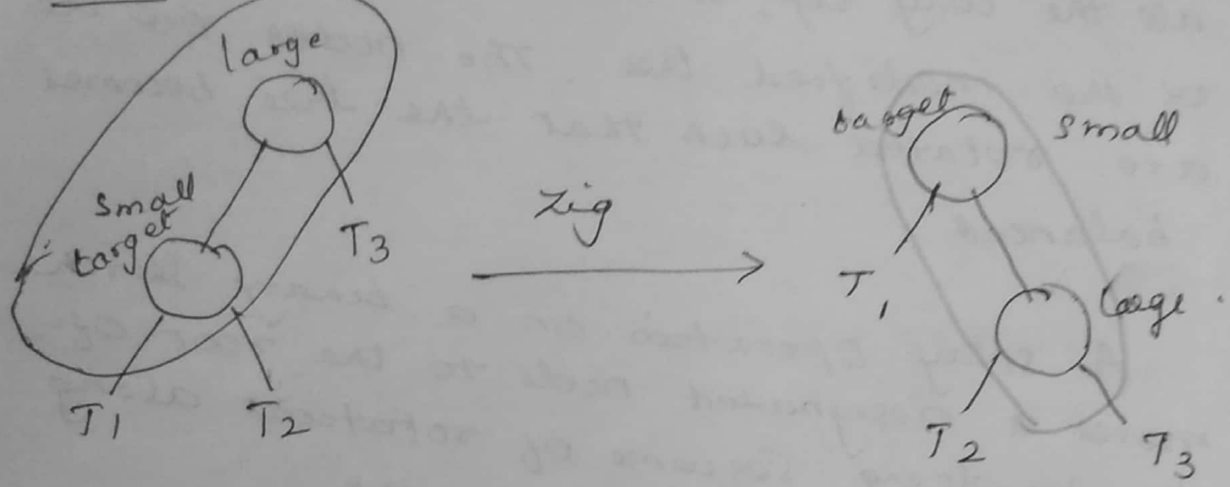
Domain usage of splay tree

1. Implementation caches, memory allocators, routers, garbage collectors, data compression, etc.

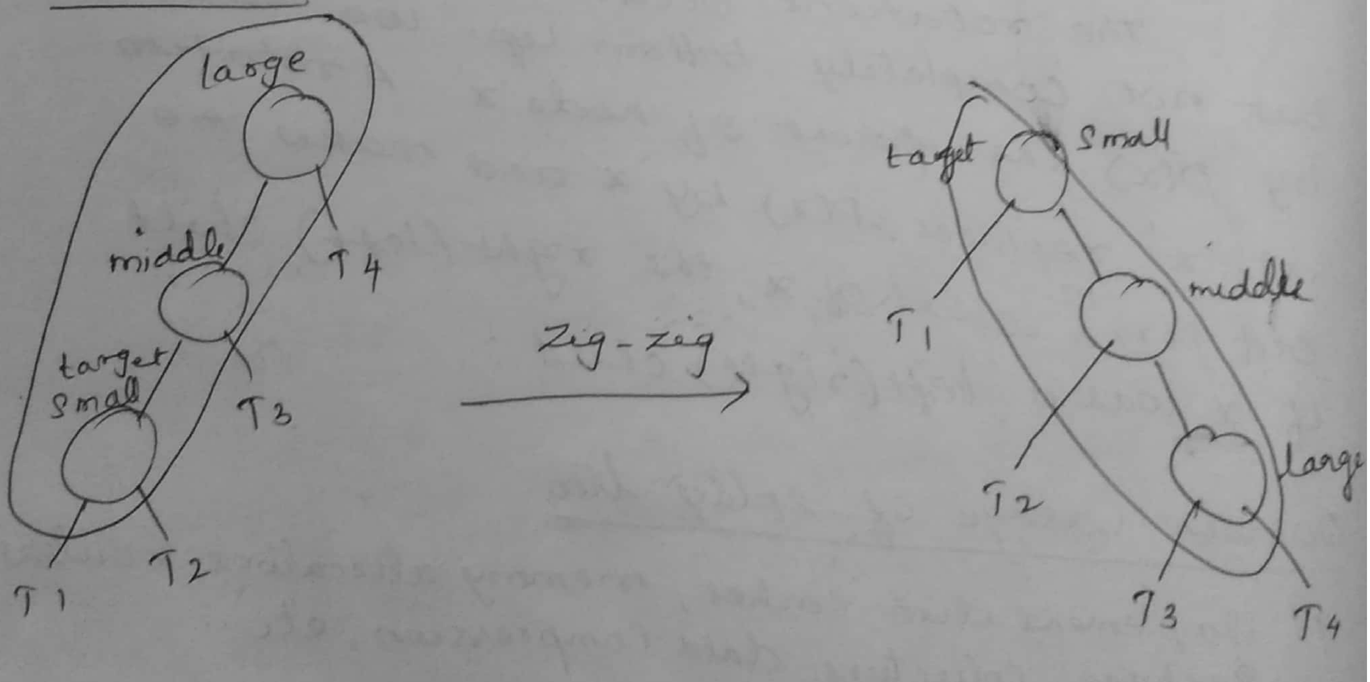
Splay step: Apply the appropriate one of the following three cases.

The rotations done in splaying for each of zig-zig, zig-zag, and zig moves are shown following figures.

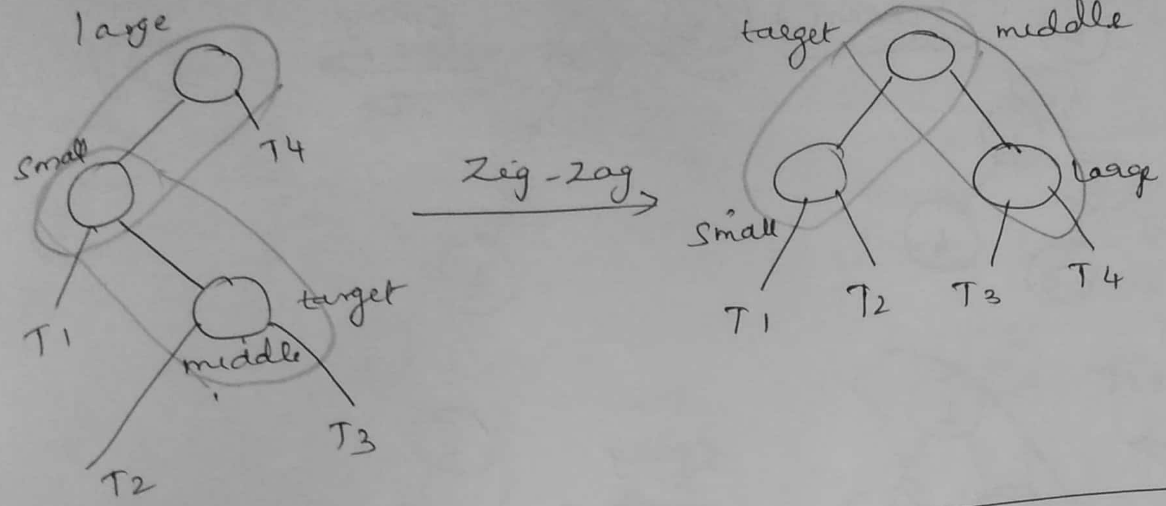
Zig



Zig-zig



Zig - Zag

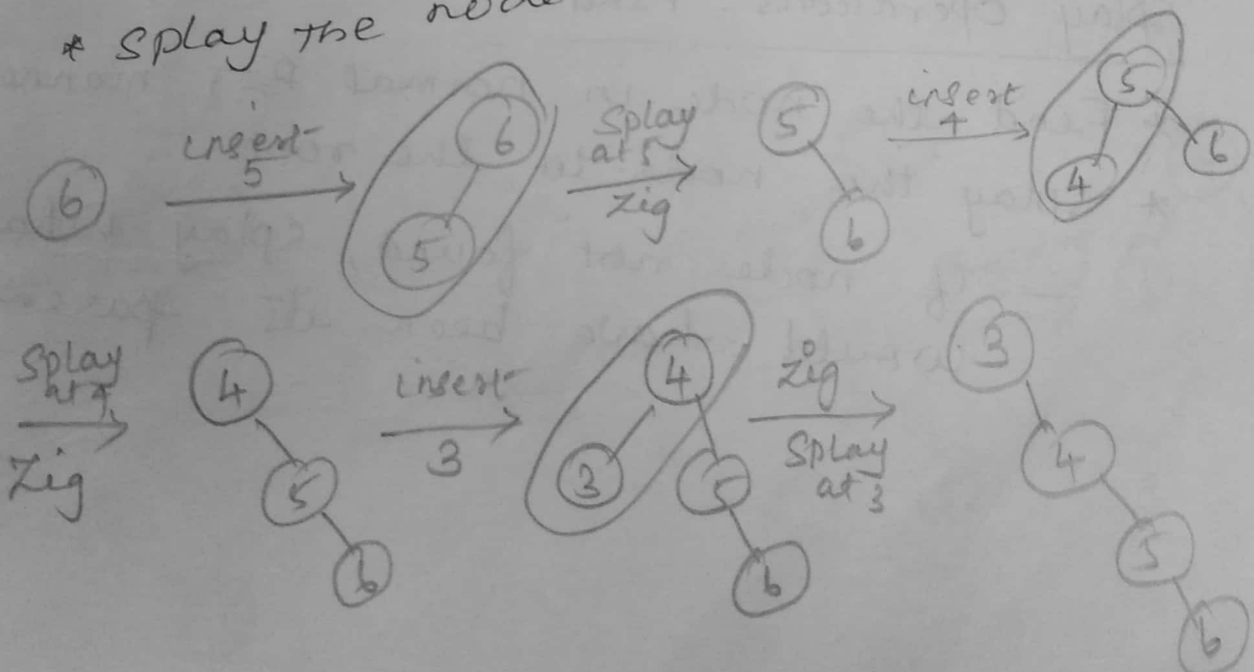


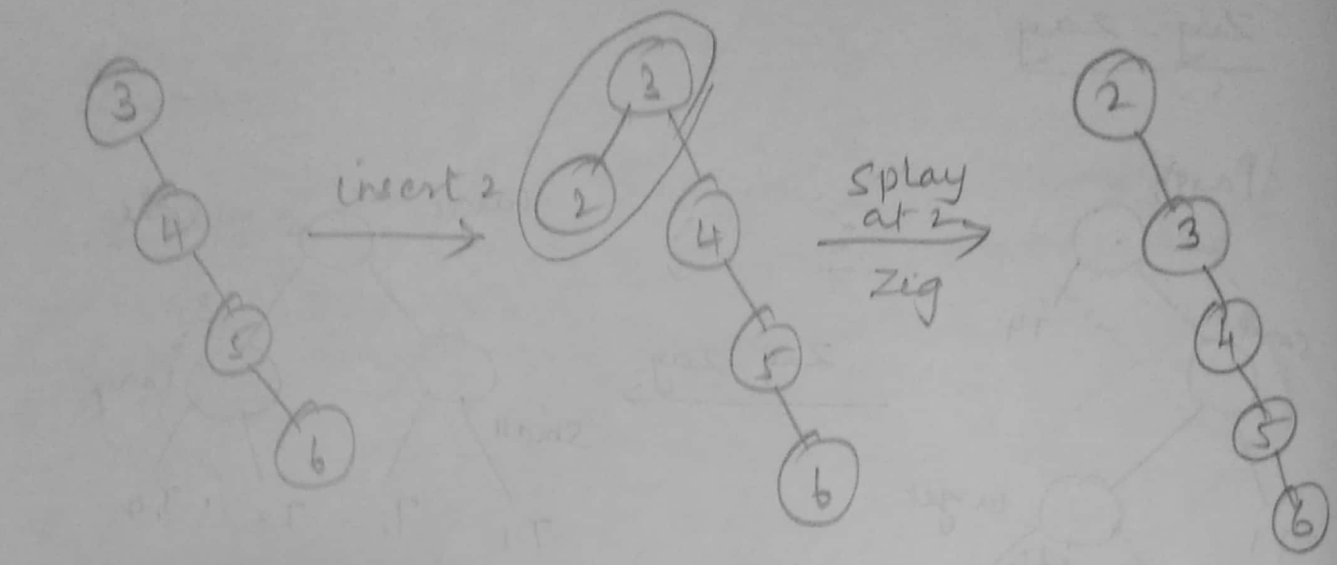
Operations on splay tree : All operations are in amortized $O(\log n)$ time

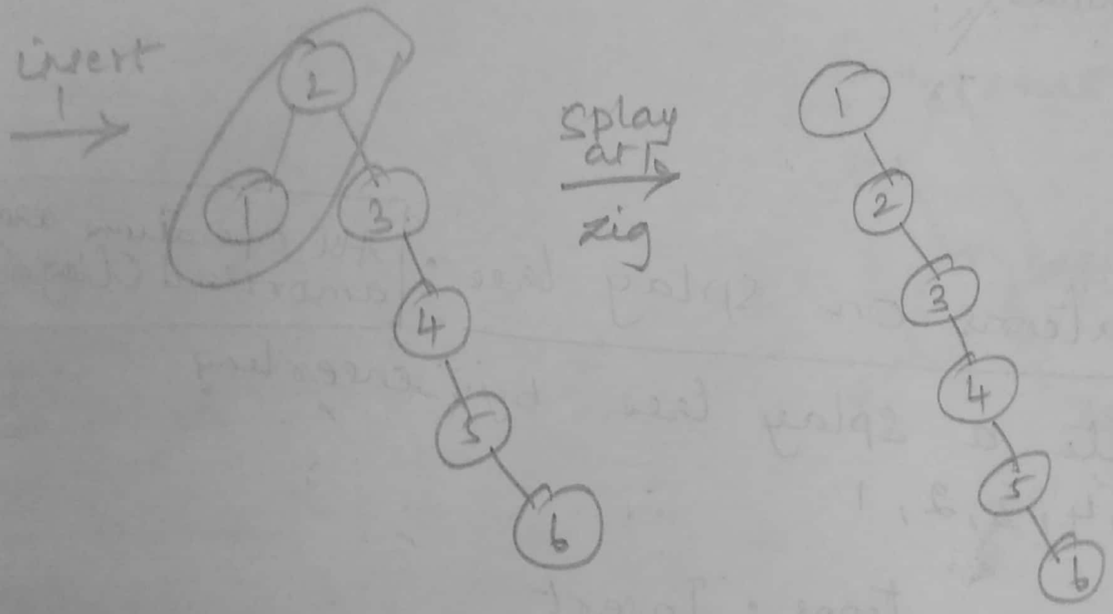
Create a splay tree by inserting
6, 5, 4, 3, 2, 1

Splay operations : Insert

* Insert the node in normal BST manner
* splay the node to the root.



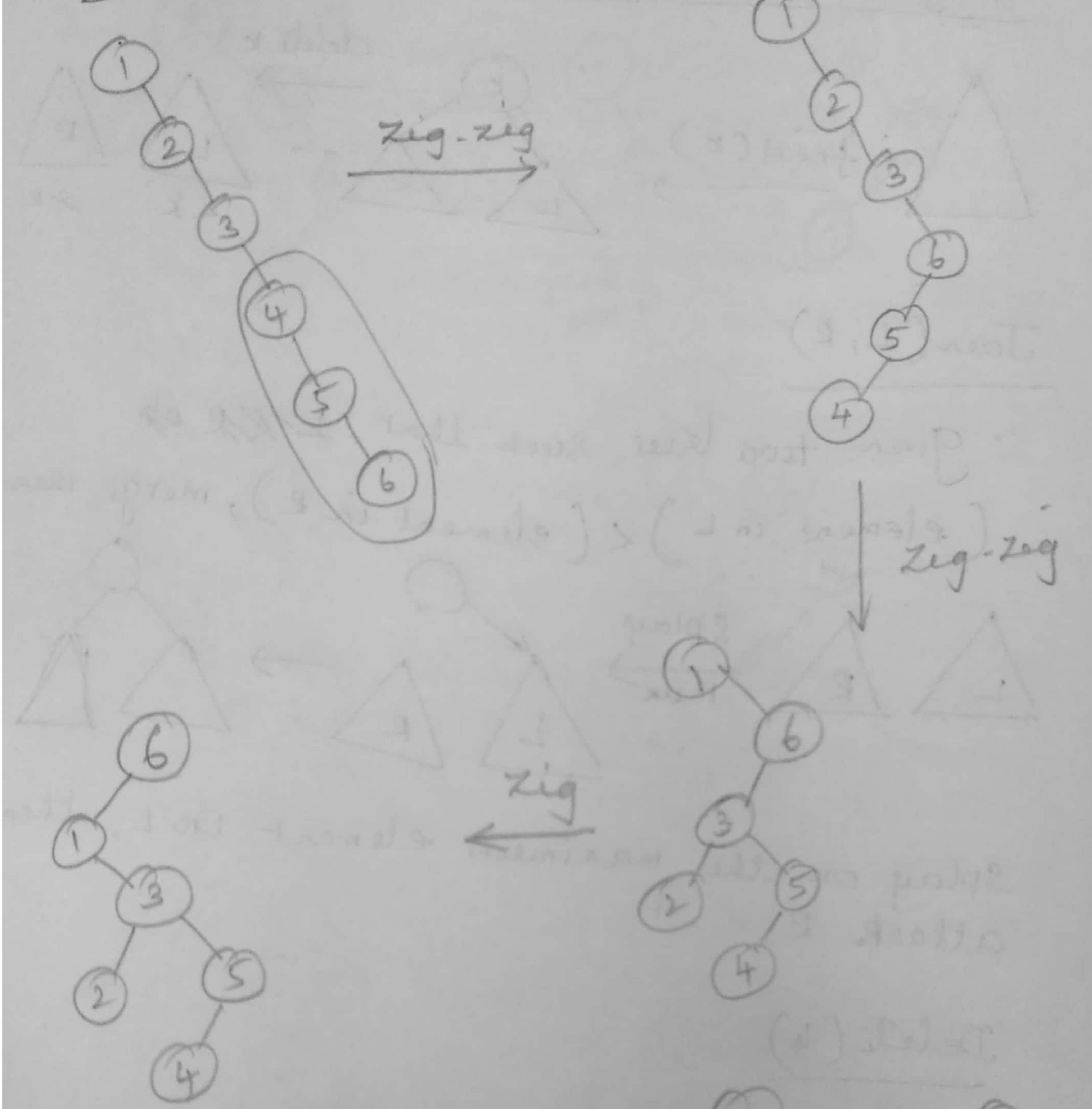




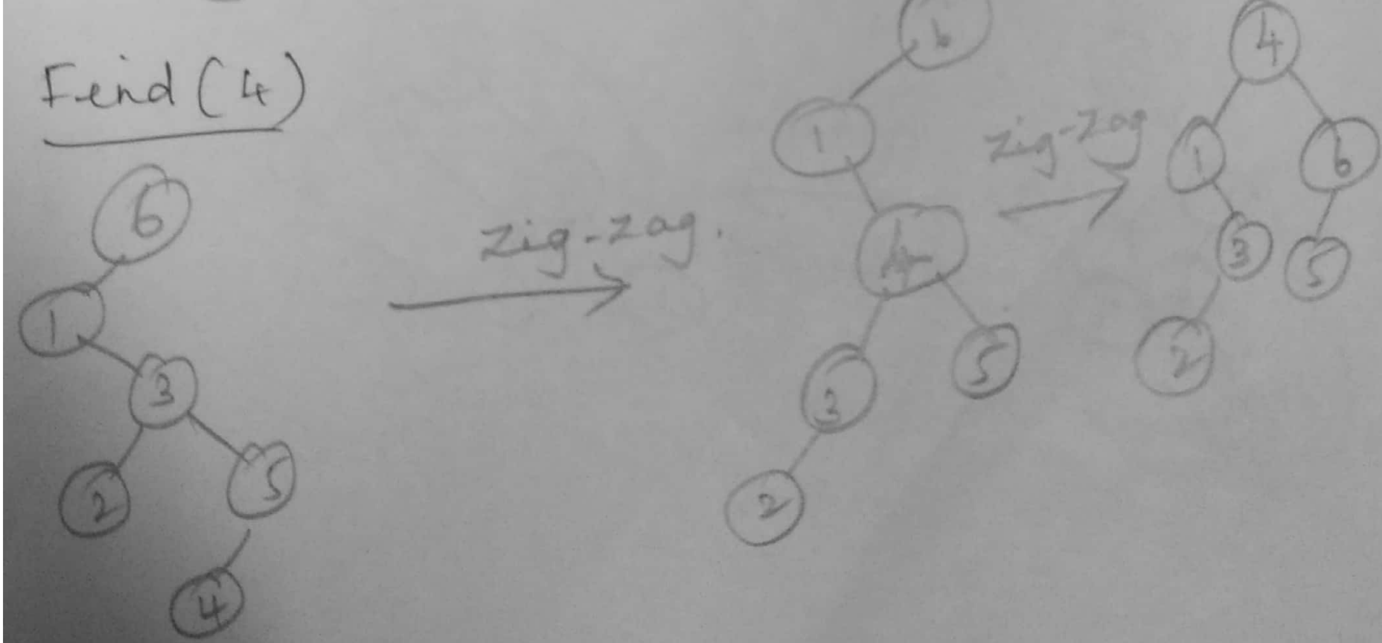
Splay operations : Find

* Find the node in normal BST manner
* Splay the node to the root.
* Splay the
  - If node not found, splay what would have been its parent.

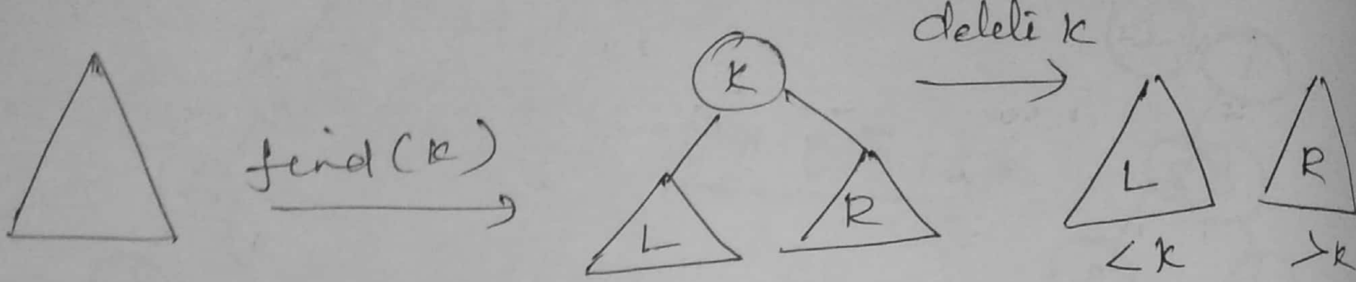
Find(6)



Find(4)

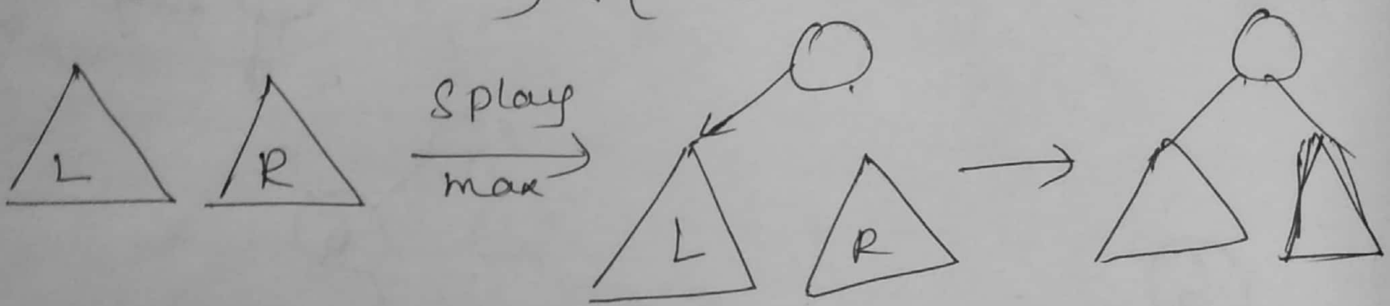


Splay operations: Remove.



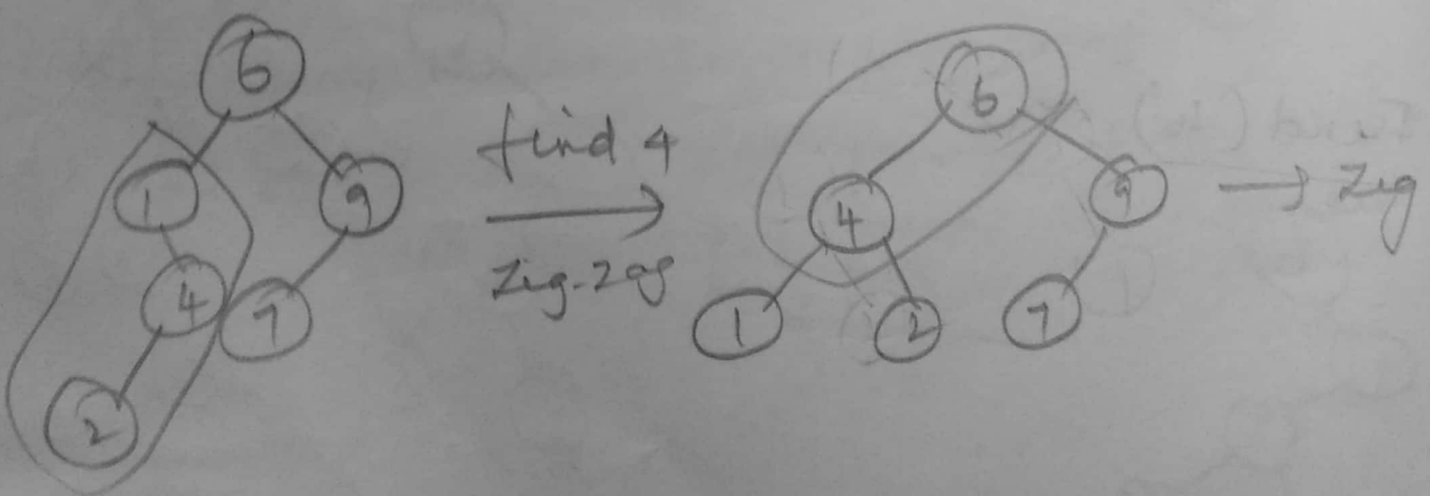
Join (L, R)

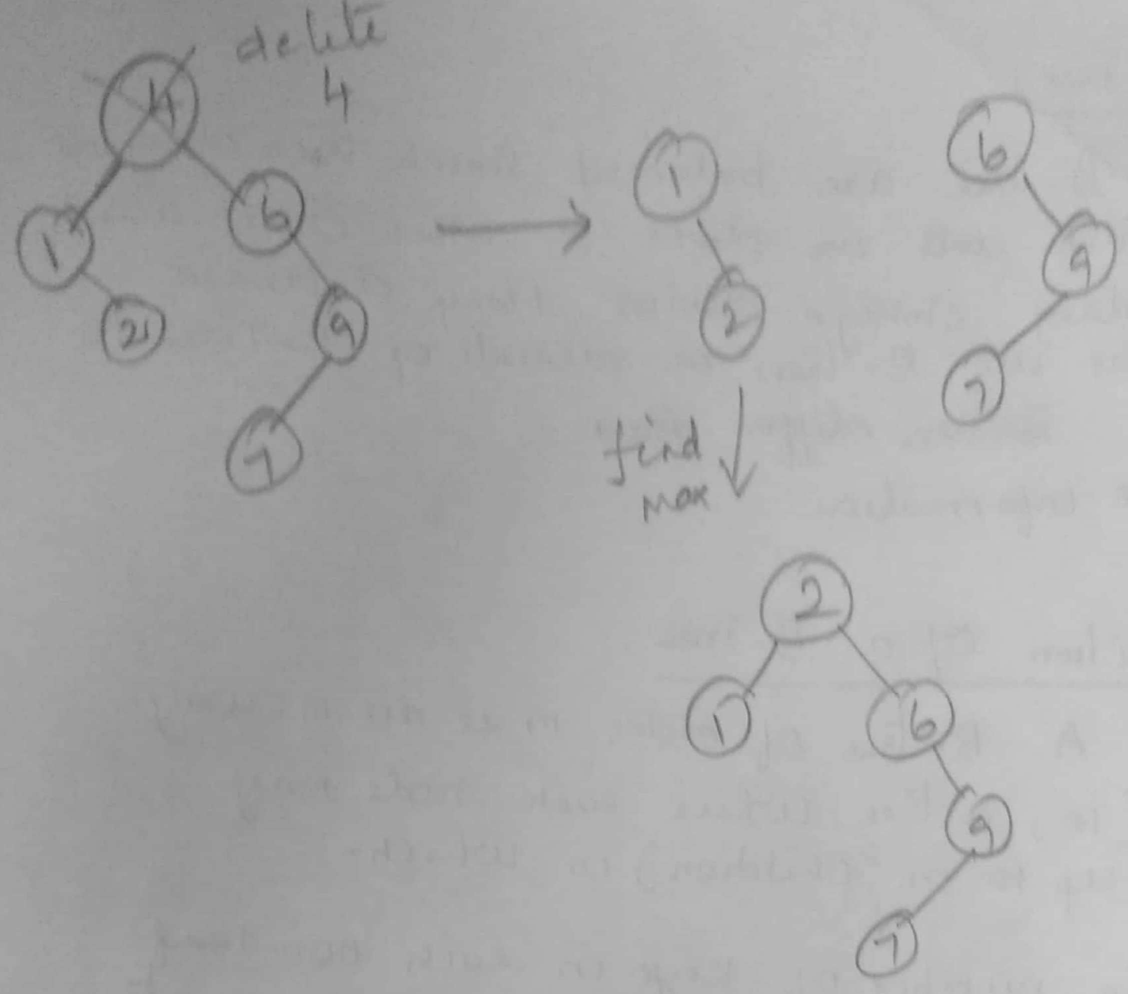
given two trees such that ~~K/K/R ek~~ (element in L) < (element in R), merge them



Splay on the maximum element in L, then attach R.

Delete (4)




delete 4
find max