1. Explain how you will analyze the performance of an algorithm. Write about the different asymptotic notations.

Ans: We generally after writing the algorithm or program need to verify whether it is efficient one or not. To know whether it is efficient or not we perform algorithm analysis. Algorithm analysis is nothing but a study to measuring the time complexity and space complexity. An algorithm is said to be efficient if it runs in minimum time and requires less main memory. Let us try to know what is time complexity and space complexity in nutshell form.

1.2.1 Time Complexity Analysis

A study conducted to calculate the amount of time required running an application or algorithm in the best case, average case and worst case is known as Time Complexity Analysis. Best case represents the minimum time, average case represents the average time and worst case represents the maximum time required to run an algorithm.

Ex:

1.2.2 Space Complexity Analysis

It is performed to measure the amount of physical memory required to run an algorithm or a program in the best case, average case and worst case.

Ex:

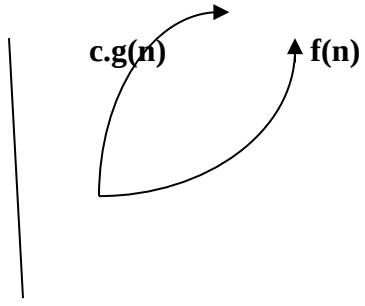
1.3 Asymptotic Notations

A variety of notations are used to represent the time complexity and space complexity of algorithms, such notations are popularly known as Asymptotic Notations. These are actually mathematical tools to represent the complexity of algorithms or programs. There are several asymptotic notations, out of which here we will try to discuss some very important notations, like Big Oh (O), Omega (Ω) and tita (ω).

1.3.1 Big Oh (O)

It is upper bound notation, always used to represent the upper bound time complexity of any algorithm or program.

Def: **f (n) = O (g(n))** iff there exist some positive constants n,n0 and c such that **f(n)<=c.g(n)** where **n > n0**



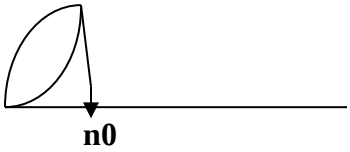


Fig 1.1: Upper Bound Time Complexity

Note: From the diagram it is always clear that f(n) is always less than c.g(n) after n0.

1.3.2 Omega (Ω)

It is lower bound notation, always used to represent the lower bound time complexity of any algorithm or program**.**

Def: f (n) = Ω (g(n)) iff there exist some positive constants n,n0 and c such that **f(n)>=c.g(n)**, where **n>n0**.

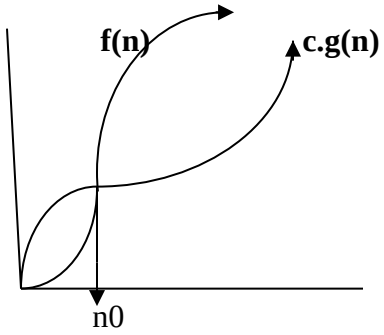


Fig 1.2: Lower Bound Time Complexity

Note: From the diagram it is clear that f(n) is always greater that c.g(n) after n0.

1.3.3 Tita (ω)

It is tight bound notation, always used to represent the average bound time complexity of an algorithm or program.

Def: f(n)= ω(g(n)) iff there exists some positive constants n,n0 and c such that **c1.g(n)<=f(n)<=g(n)** where **n>n0**.

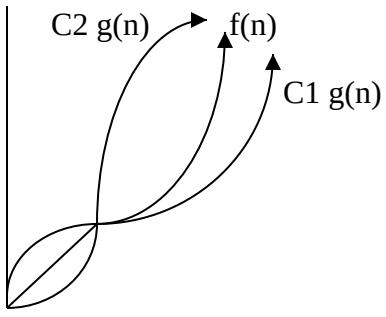


Fig 1.3: Average Bound Time Complexity

Note: Finding an average bound of any algorithm or function is not possible.

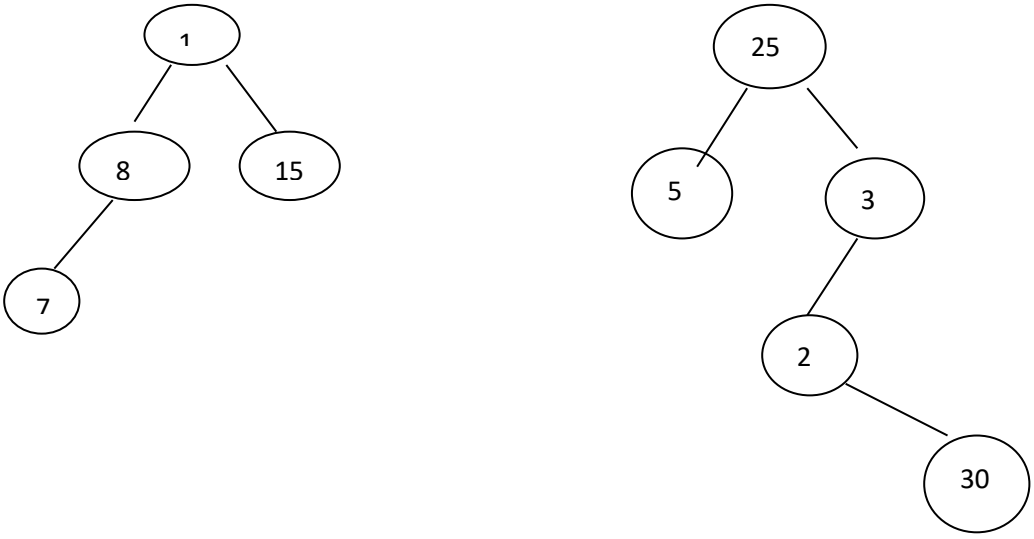
There are three methods to solve recurrences. They are substation method, master's method and recurrence tree method.

2. Write a brief note on AVL trees and Priority queues.

Ans:

An AVL tree is self height balanced binary search tree in which the height difference between the left sub tree and right sub tree is at most differed by 1. AVL tree is developed by **A**delson **V**elski **L**andid so it is named as AVL.

Examples:



In the above figures a1 is an AVL tree and a2 is not an AVL tree.

If the height difference between the left sub tree and right sub tree is greater than one then tree is balanced using some rotations. They are as follows

1. Left rotation
2. Right rotation
3. Left Right rotation
4. Right Left rotation

Left Rotation: A left rotation is required if the BST is right heavy tree. A right heavy tree means the height of the right sub tree is more than the left sub tree. Example for right heavy BST is R1. To know whether a BST is right heavy or not we calculate the balance factor. The balance factor is calculated as follows.

Balance factor = height of the right sub tree – height of the left sub tree.

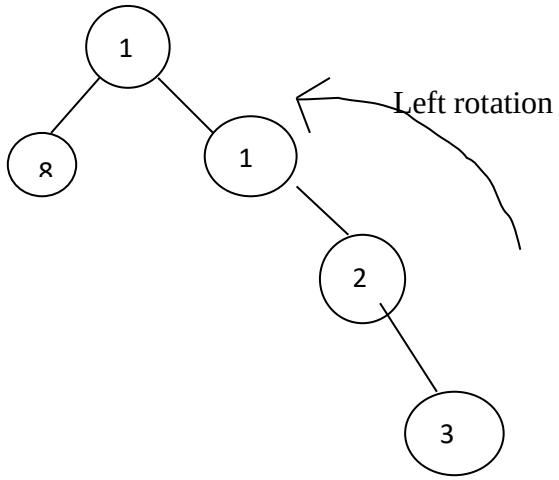


Fig 5.34

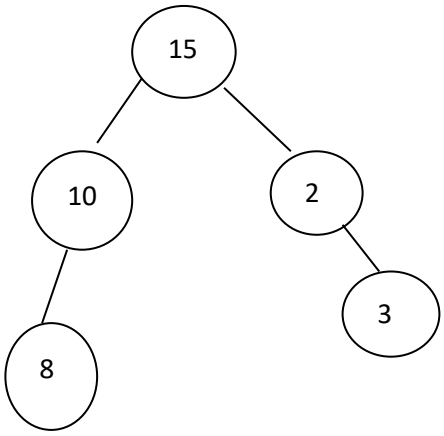
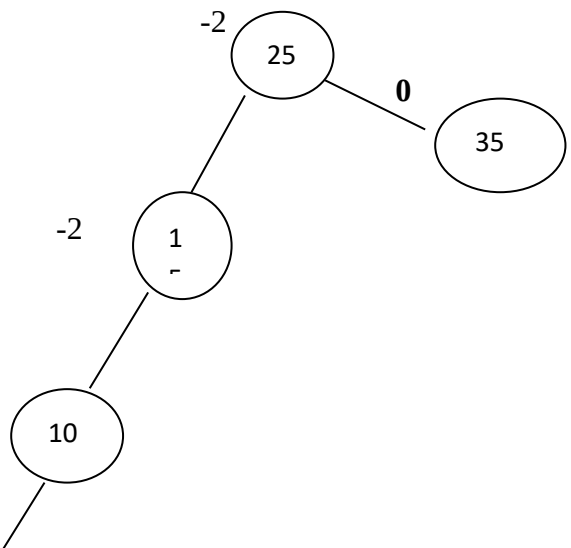


Fig 5.35

As the tree R1 is right heavy BST we need to perform a left rotation. The resultant BST after left rotation we can see in the figure R2.

Right rotation: A right rotation is required when the given tree is left heavy. A left heavy tree is a BST in which the height of the left sub tree is more than the right sub tree. To know whether the tree is left heavy or not we calculate the balance factor. If the balance factor of any node is less than -1, we can assume that the tree is a left heavy tree. In such situations we perform right rotation on the node where the balance factor is less than -1to make the tree balanced.



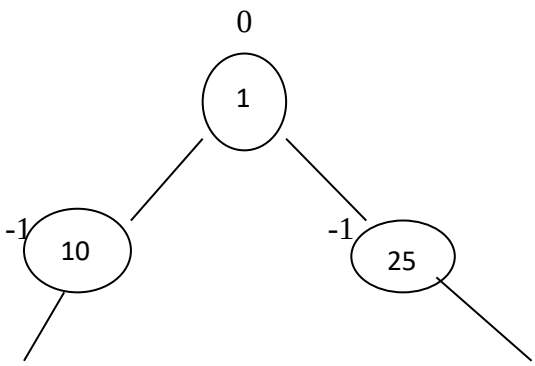


Fig 5.36 (UN balanced AVL tree)

Fig 5.37 (Balanced AVL tree)

Left right rotation: It is a combination of left rotation followed by a right rotation. If the balance of the AVL tree is disturbed by inserting some element in the left sub tree as a right child then we to perform a left rotation followed by a right rotation.

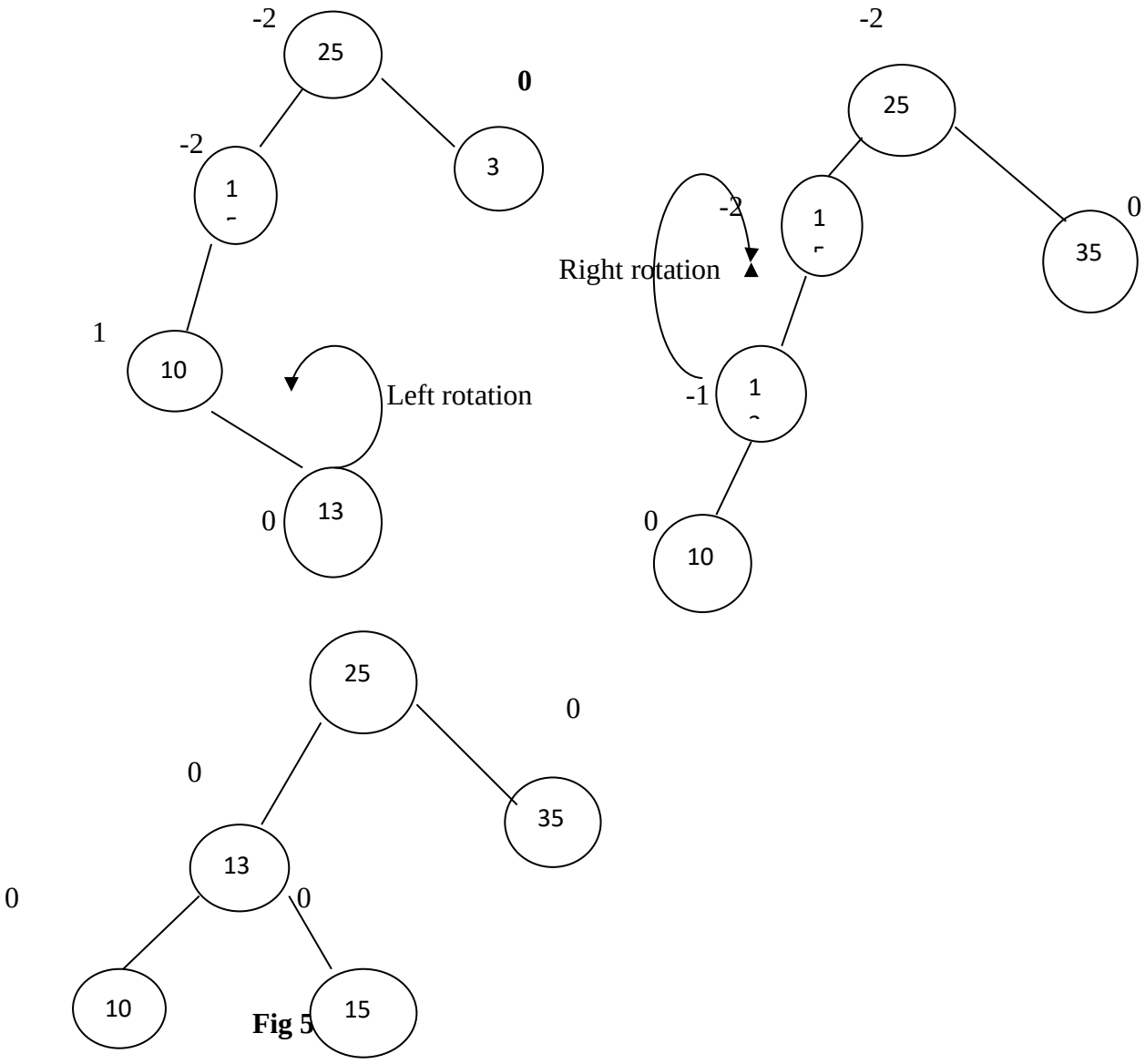


Fig 5

Right Left Rotation: In right left rotation we perform right rotation followed by left rotation. Right left rotation is required when we insert some element in the right sub tree as a left child and assume because of this insertion the height of the AVL tree is disturbed.

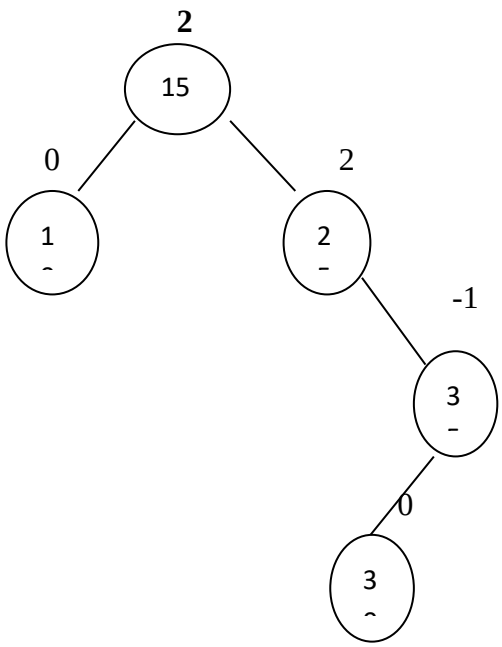


Fig 5.41

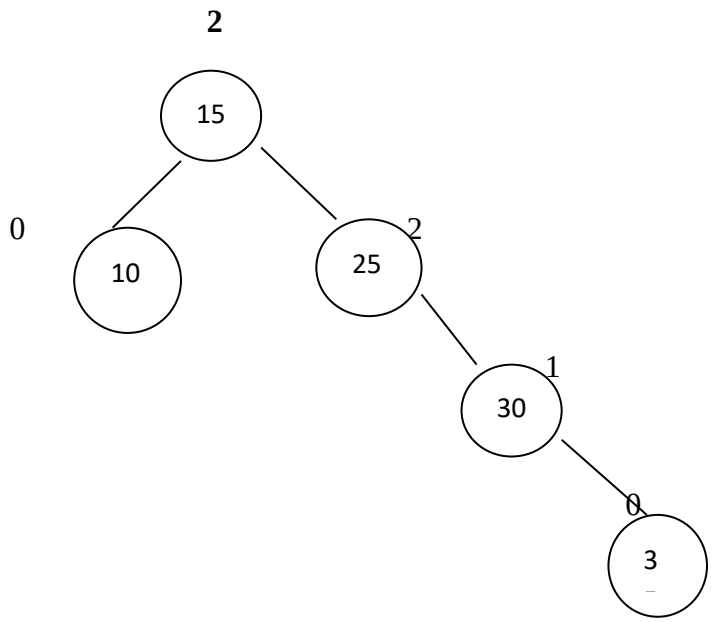


Fig 5.42

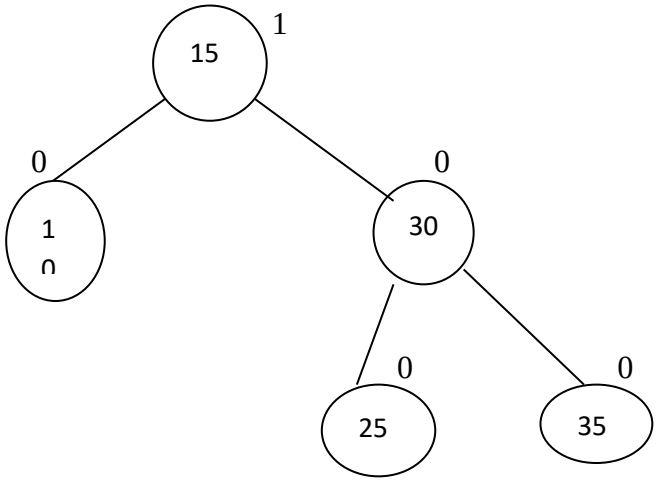
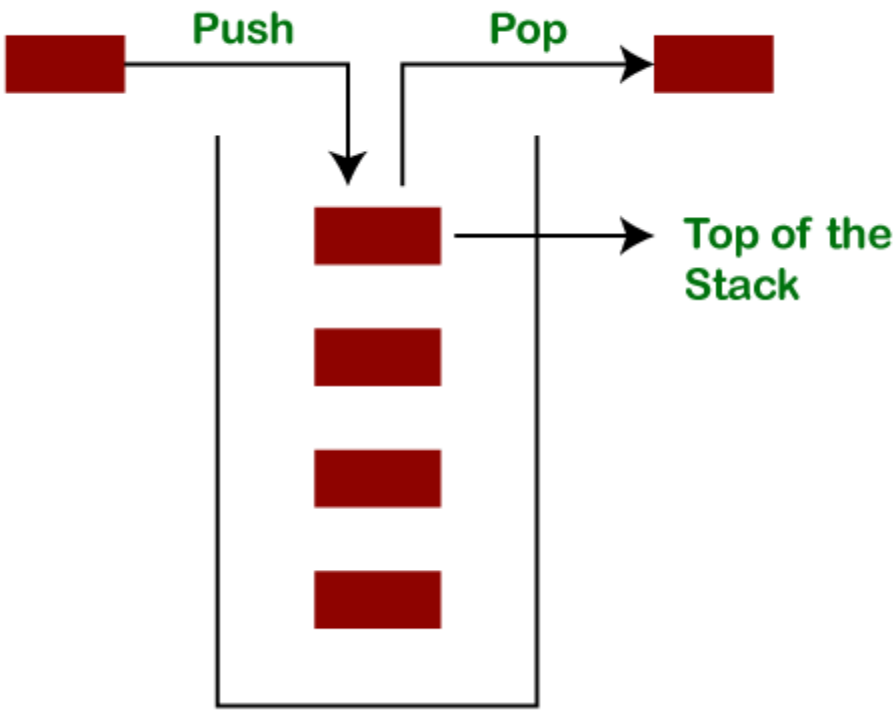


Fig 5.43

3.Define Stack. Write the pseudo code for the stack operations.



Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

There are many real-life examples of a stack. Consider an example of plates stacked over one another in the canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO(Last In First Out)/FILO(First In Last Out) order.

Mainly the following four basic operations are performed in the stack:

Push: Adds an item to the stack. If the stack is full, then it is said to be an Overflow condition.

Algorithm for push:

```
begin
 if stack is full
    return
 endif
else
 increment top
 stack[top] assign value
end else
```

```
end procedure
```

Pop: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
Algorithm for pop:

```
begin
 if stack is empty
    return
 endif
else
 store value of stack[top]
 decrement top
 return value
end else
end procedure
```

Peek or Top: Returns the top element of the stack.
Algorithm for peek:

```
begin
  return stack[top]
end procedure
```

isEmpty: Returns true if the stack is empty, else false.
Algorithm for isEmpty:

```
begin
 if top < 1
    return true
 else
    return false
end procedure
```

4.Explain the queue linear data structures. Why the linked list representation of the queue is efficient than the array representation justify.

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −

As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue −

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows −

Algorithm

```
begin procedure peek
return queue[front]
end procedure
```

Implementation of peek() function in C programming language −

Example

```
int peek() {
```

```
   return queue[front];
}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

Algorithm

```
begin procedure isfull

   if rear equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

Implementation of isfull() function in C programming language −

Example

```
bool isfull() {
   if(rear == MAXSIZE - 1)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function −

Algorithm

```
begin procedure isempty

   if front is less than MIN  OR front is greater than rear
      return true
   else
      return false
   endif

end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code −

Example

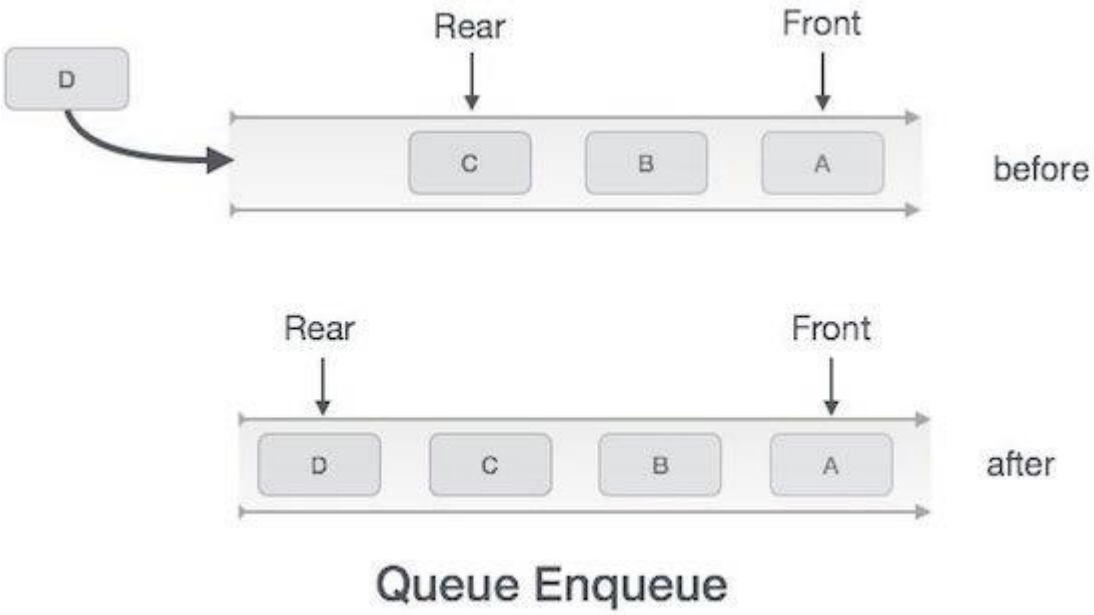
```
bool isempty() {
   if(front < 0 || front > rear)
      return true;
   else
      return false;
}
```

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success.



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation

```
procedure enqueue(data)

   if queue is full
     return overflow
   endif

   rear ← rear + 1
   queue[rear] ← data
   return true

end procedure
```

Implementation of enqueue() in C programming language −

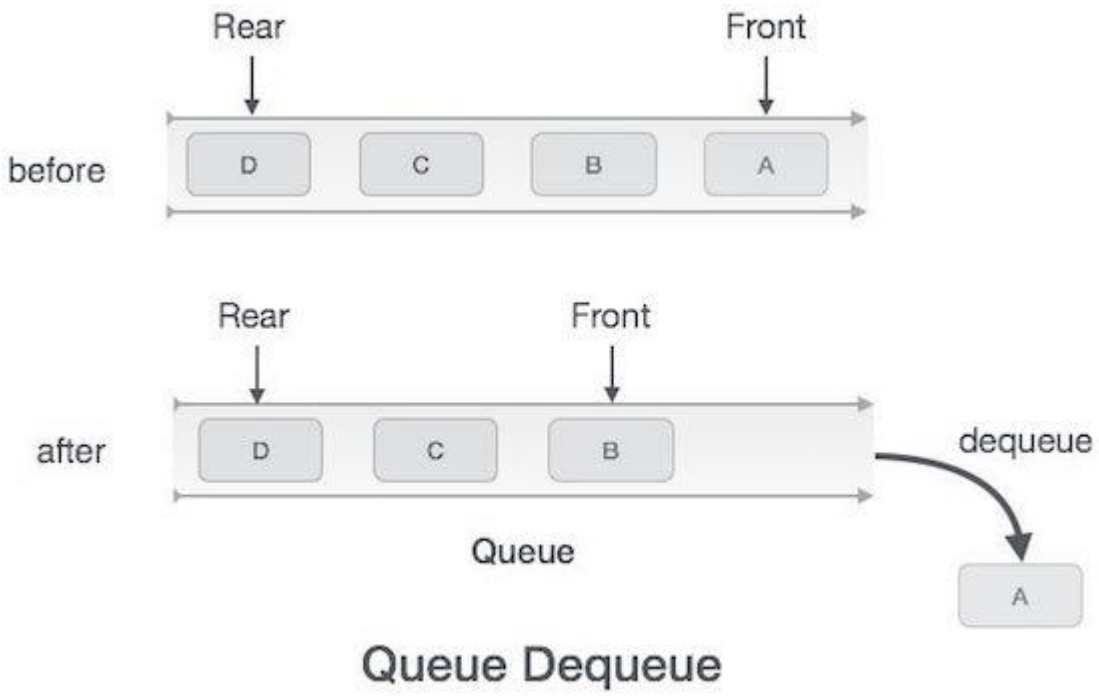
Example

```
int enqueue(int data)
   if(isfull())
      return 0;

   rear = rear + 1;
   queue[rear] = data;

   return 1;
end procedure
```

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation −

- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return success.



Queue Dequeue

Algorithm for dequeue operation

```
procedure dequeue

   if queue is empty
      return underflow
   end if

   data = queue[front]
   front ← front + 1
   return true

end procedure
```

Implementation of dequeue() in C programming language −

Example

```
int dequeue() {
   if(isempty())
      return 0;

   int data = queue[front];
   front = front + 1;

   return data;
}
```

the array implementation can not be used for the large scale applications where the queues are implemented. One of the alternative of array implementation is linked list implementation of queue.

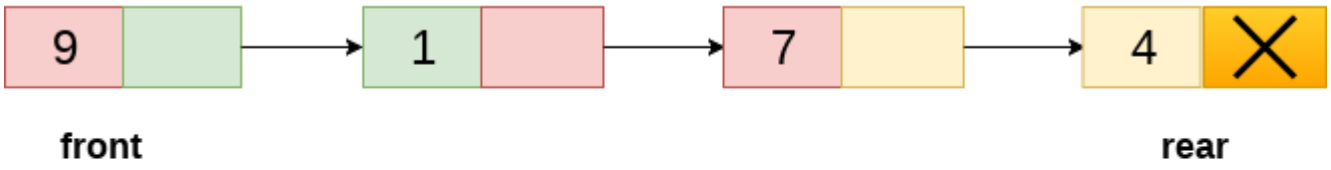
The storage requirement of linked representation of a queue with n elements is o(n) while the time requirement for operations is o(1).

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

The linked representation of queue is shown in the following figure.



Linked Queue

Operation on Linked Queue

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.

Insert operation

The insert operation append the queue by adding an element to the end of the queue. The new element will be the last element of the queue.

Firstly, allocate the memory for the new node ptr by using the following statement.

```
Ptr = (struct node *) malloc (sizeof(struct node));
```

There can be the two scenario of inserting this new node ptr into the linked queue.

In the first scenario, we insert element into an empty queue. In this case, the condition **front = NULL** becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL.

```
ptr -> data = item;
    if(front == NULL)
    {
        front = ptr;
        rear = ptr;
        front -> next = NULL;
        rear -> next = NULL;
    }
```

In the second case, the queue contains more than one element. The condition front = NULL becomes false. In this scenario, we need to update the end pointer rear so that the next pointer of rear will point to the new node ptr. Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node **ptr**. We also need to make the next pointer of rear point to NULL.

5. Define an algorithm. Explain any six properties to specify an algorithm.

Algorithm

We know to solve some problem or to write a program, generally follow some sequence of steps. In simple terms we can define an algorithm as a step by step procedure for problem solving. To show how an algorithm is generally written let us consider the following example.

Algorithm: Add(X, Y, Z)
Step 1: Read the first number
X<-First Number
Step 2: Read the second number
Y<- Second Number
Step3: Add X to Y and Store the result into Z
Z<-X+Y
Step4: Stop

Algorithms generally have the following characteristics:

1. **Input:** The algorithm receives input. Zero or more quantities are externally supplied.
2. **Output:** The algorithm produces output. At least one quantity is produced.
3. **Precision:** The steps are precisely stated. Each instruction is clear and unambiguous.
4. **Feasibility:** It must be feasible to execute each instruction.
5. **Flexibility:** It should also be possible to make changes in the algorithm without putting so much effort on it.
6. **Generality:** The algorithm applies to a set of inputs.
7. **Finiteness:** Algorithm must complete after a finite number of instruction have been executed.

6. Design an algorithm to find the largest of a given list of items and analyze its efficiency.

Given an array **arr** of size **N**, the task is to find the largest element in the given array.
Example:
Input: *arr[] = {10, 20, 4}*
Output: *20*
Input : *arr[] = {20, 10, 20, 4, 100}*
Output : *100*

Linear Traversal: One of the most simplest and basic approach to solve this problem is to simply traverse the whole list and find the maximum among them.
Follow the steps below to implement this idea:

- Create a local variable **max** to store the maximum among the list
- **Initialize max with the first element** initially, to start the comparison.
- Then **traverse the given array** from second element till end, and for each element:
    - **Compare the current element with max**
    - If the current element is greater than max, then **replace** the value of **max** with the current element.
- At the end, **return** and print the value of the largest element of array stored in **max**.

Time complexity: O(N), to traverse the Array completely.
Auxiliary Space: O(1), as only an extra variable is created, which will take O(1) space.

7. Derive Strassen's matrix multiplication time complexity using substitution method.

   Ans: Given two square matrices A and B of size n x n each, find their multiplication matrix.

   Naive Method

   Following is a simple way to multiply two matrices.

Strassen's has proposed an algorithm for multiplication of two 2*2 matrices which requires only 7 recursive calls. So using strassen's approach we can reduce the time complexity.

The general approach for matrix multiplication based on divide and conquers approach is as follows.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad \begin{bmatrix} e & f \\ g & h \end{bmatrix} \quad = \quad \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

In divide and conquer technique to multiply two 2*2 matrices we will divide the matrix into 4 1*1 matrices and we use the general procedure for the matrix multiplication.

In strassens matrix multiplication approach we use the following procedure for matrix multiplication.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad \begin{bmatrix} e & f \\ g & h \end{bmatrix} \quad = \quad \begin{bmatrix} p5+p4-p2+p6 & p1+p2 \\ p3+p4 & p1+p5-p3-p7 \end{bmatrix}$$

p1=a(f-h), p2=(a+b)h, p3=(c+d)e, p4=d(g-e)

p5=(a+d)(e+h), p6=(b-d)(g+h), p7=(a-c)(e+f)

Strassen's matrix multiplication requires only 7 recursions where as naïve method and simple divide and conquer approach needs 8 recursions.

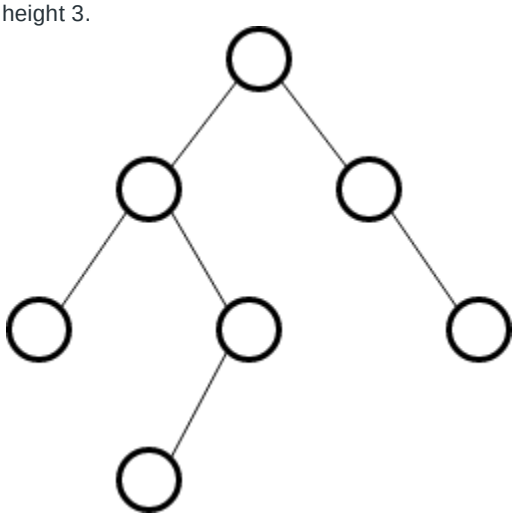
Time Complexity: T (N) = 7T (N/2) + O (N*N)

Generally we use some other algorithms for matrix multiplication because of some reasons like so many constants are used in strassen's algorithm and too many errors.

8. What is the maximum height of any AVL-tree with 7 nodes? Assume that the height of a tree with a single node is 0.

Solution: For finding maximum height, the nodes should be minimum at each level. Assuming height as 2, minimum number of nodes required:
N(h) = N(h-1) + N(h-2) + 1
N(2) = N(1) + N(0) + 1 = 2 + 1 + 1 = 4.
It means, height 2 is achieved using minimum 4 nodes.
Assuming height as 3, minimum number of nodes required:
N(h) = N(h-1) + N(h-2) + 1
N(3) = N(2) + N(1) + 1 = 4 + 2 + 1 = 7.
It means, height 3 is achieved using minimum 7 nodes.

Therefore, using 7 nodes, we can achieve maximum height as 3. Following is the AVL tree with 7 nodes and height 3.



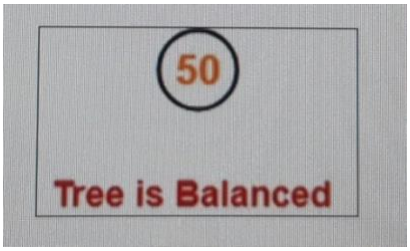
10. Construct AVL Tree for the following sequence of numbers -

50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48

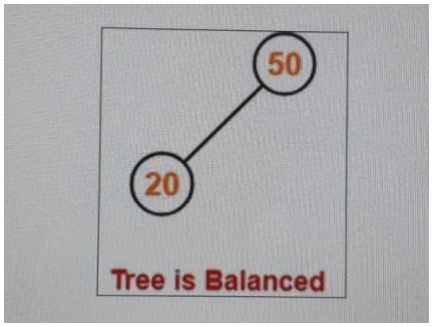
Example 1 : Construct AVL Tree for the following sequence of numbers -

50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48

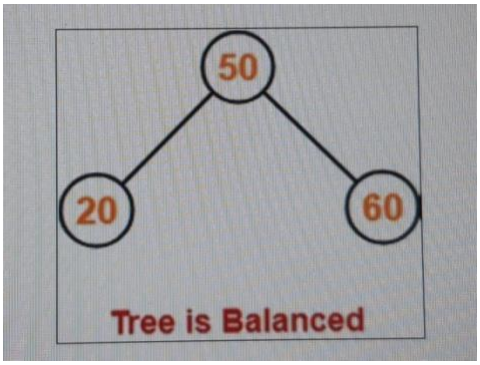
Step 1 : Insert 50



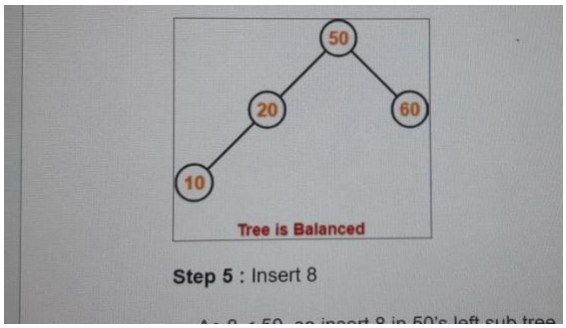
Step 2 : Insert 20 , As 20 < 50, so insert 20 in 50's left sub tree



Step 3 : Insert 60 , As 60 > 50, so insert 60 in 50's right sub tree.

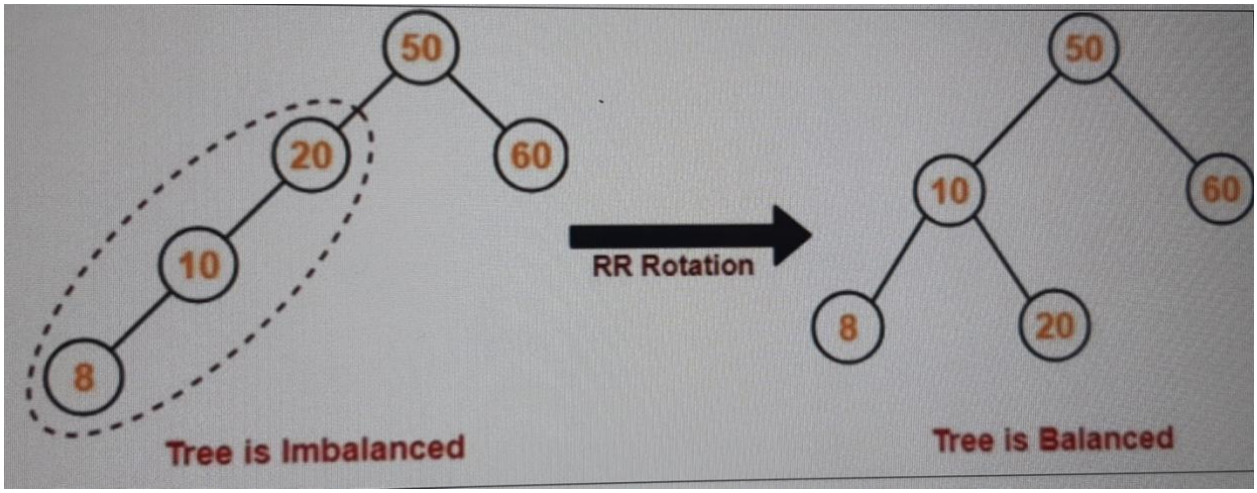


Step 4 : Insert 10



- As 10 < 50, so insert 10 in 50's left sub tree.
- As 10 < 20, so insert 10 in 20's left sub tree.

Step 5 : Insert 8



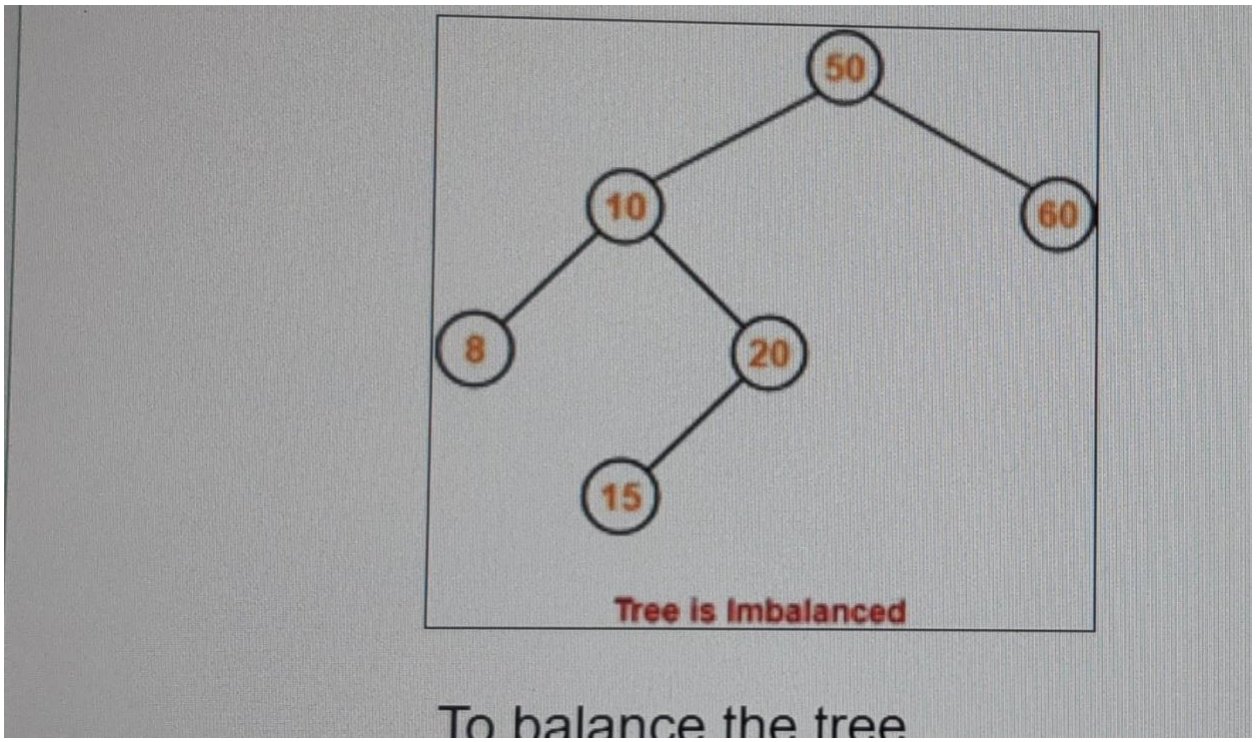
- As 8 < 50, so insert 8 in 50's left sub tree.
- As 8 < 20, so insert 8 in 20's left sub tree.
- As 8 < 10, so insert 8 in 10's left sub tree.

To balance the tree,

- Find the first imbalanced node on the path from the newly inserted node (node 8) to the root node.
- The first imbalanced node is node 20.
- Now, count three nodes from node 20 in the direction of leaf node.
- Then, use AVL tree rotation to balance the tree.

Following this, we have-

Step 6 : Insert 15

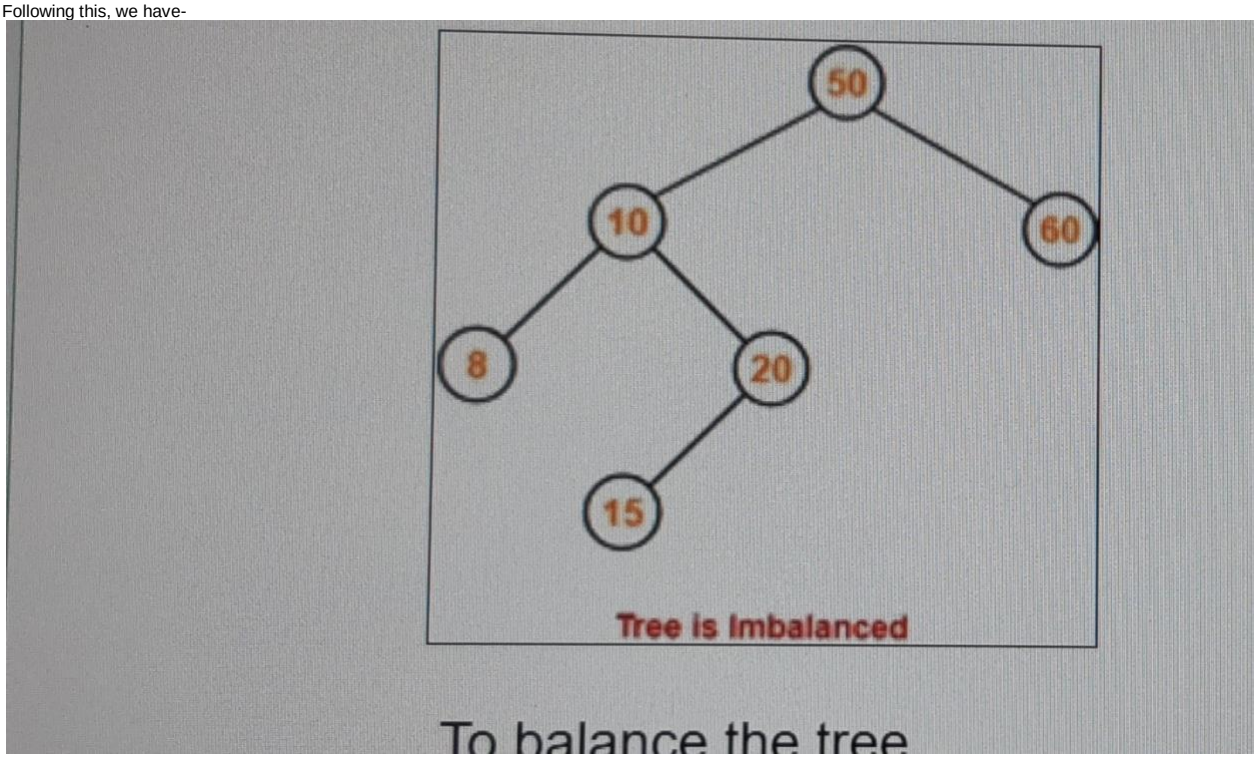


- As 15 < 50, so insert 15 in 50's left sub tree.
- As 15 > 10, so insert 15 in 10's right sub tree.
- As 15 < 20, so insert 15 in 20's left sub tree.

To balance the tree,

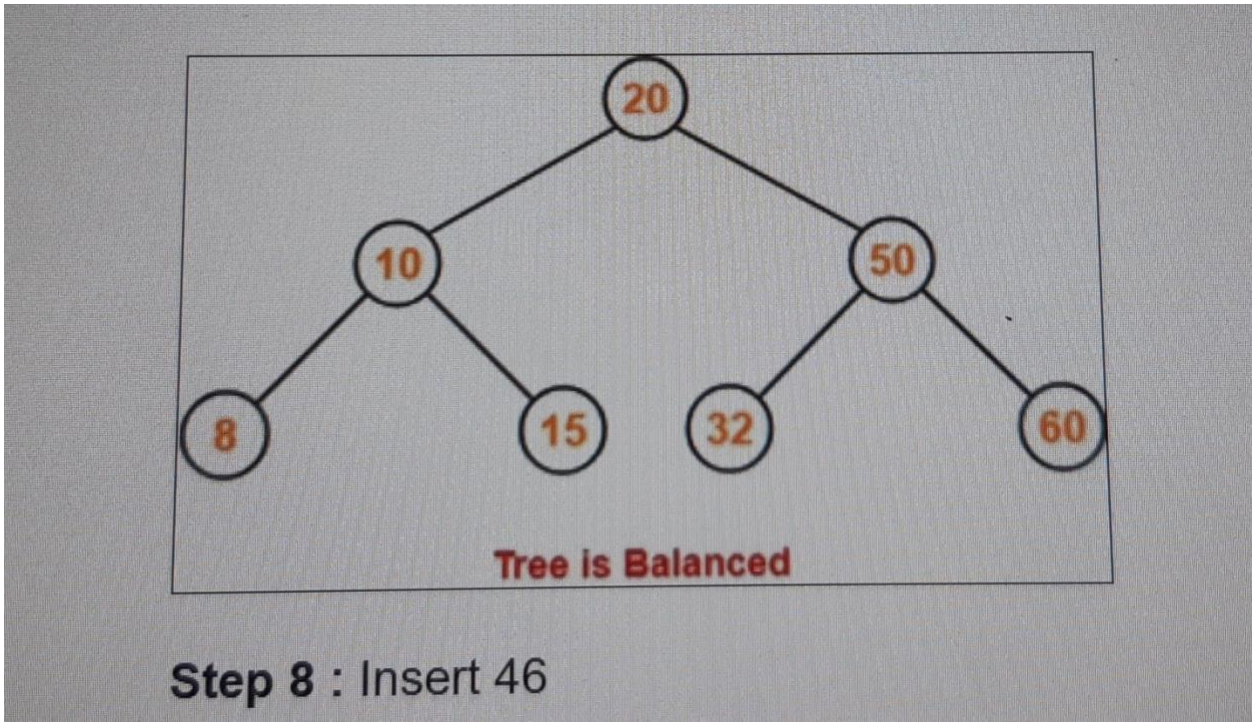
- Find the first imbalanced node on the path from the newly inserted node (node 15) to the root node.
- The first imbalanced node is node 50.
- Now, count three nodes from node 50 in the direction of leaf node.
- Then, use AVL tree rotation to balance the tree.

Following this, we have-



Step 7 : Insert 32

- As 32 > 20, so insert 32 in 20's right sub tree.
- As 32 < 50, so insert 32 in 50's left sub tree.

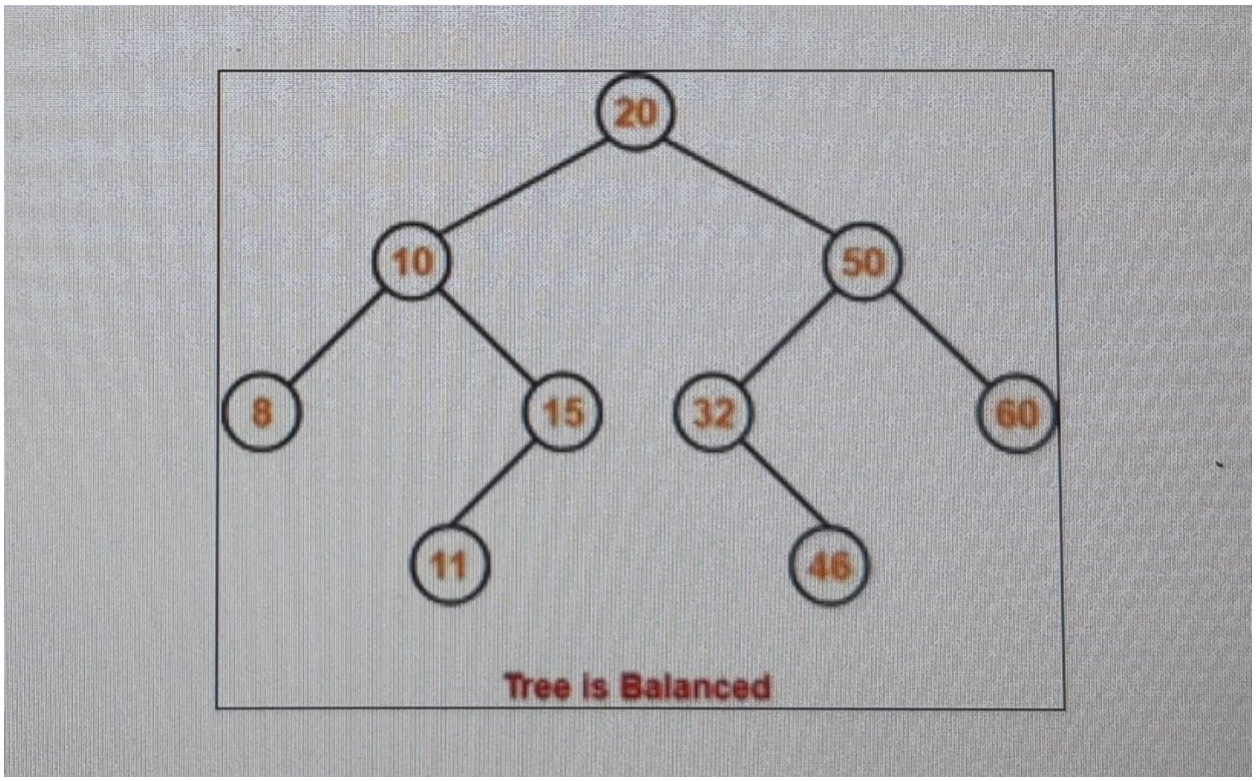


Step 8 : Insert 46

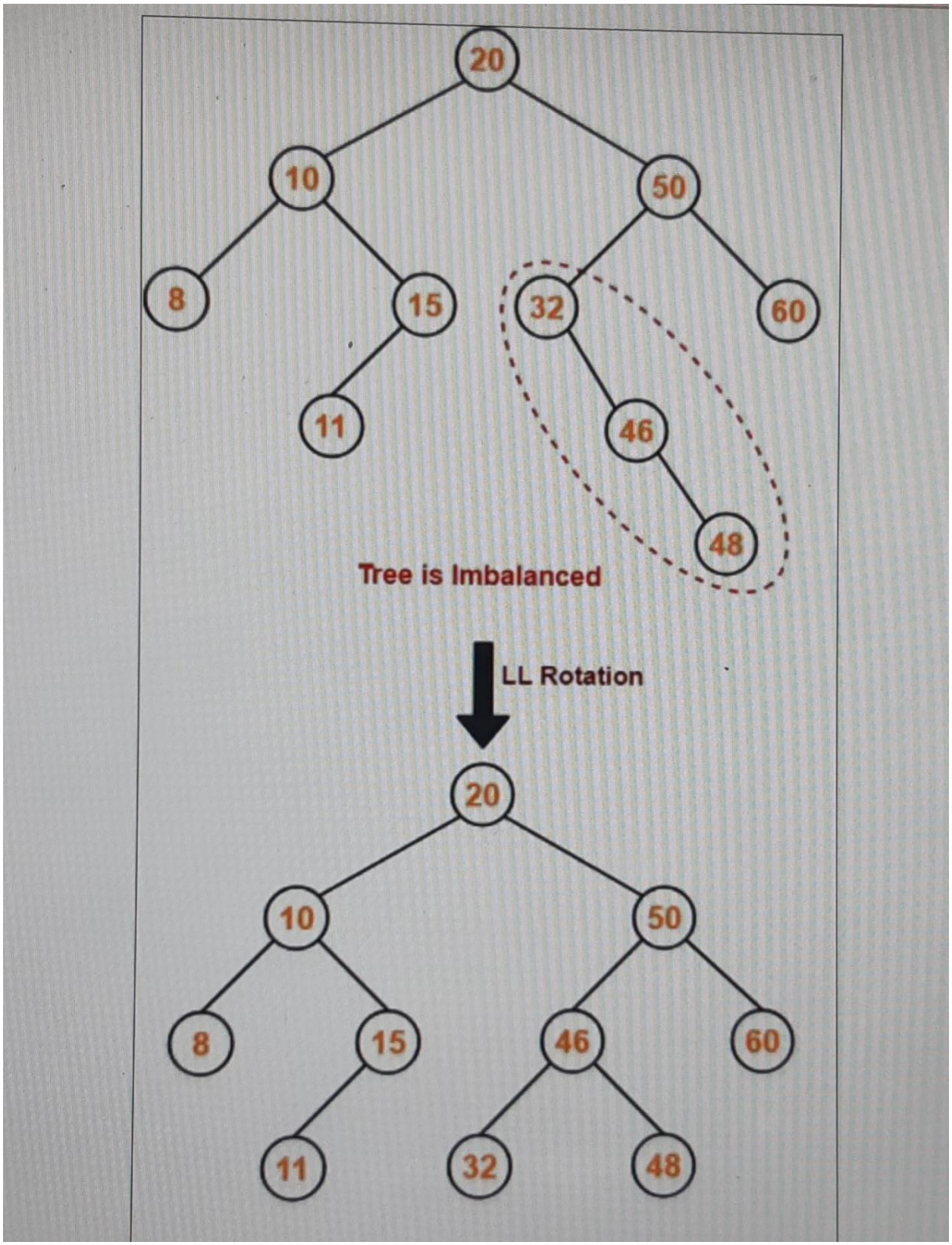
- As 46 > 20, so insert 46 in 20's right sub tree.
- As 46 < 50, so insert 46 in 50's left sub tree.
- As 46 > 32, so insert 46 in 32's right sub tree.
-

Step 9 : Insert 11

- As 11 < 20, so insert 11 in 20's left sub tree.
- As 11 > 10, so insert 11 in 10's right sub tree.
- As 11 < 15, so insert 11 in 15's left sub tree.



Step 10 : Insert 48



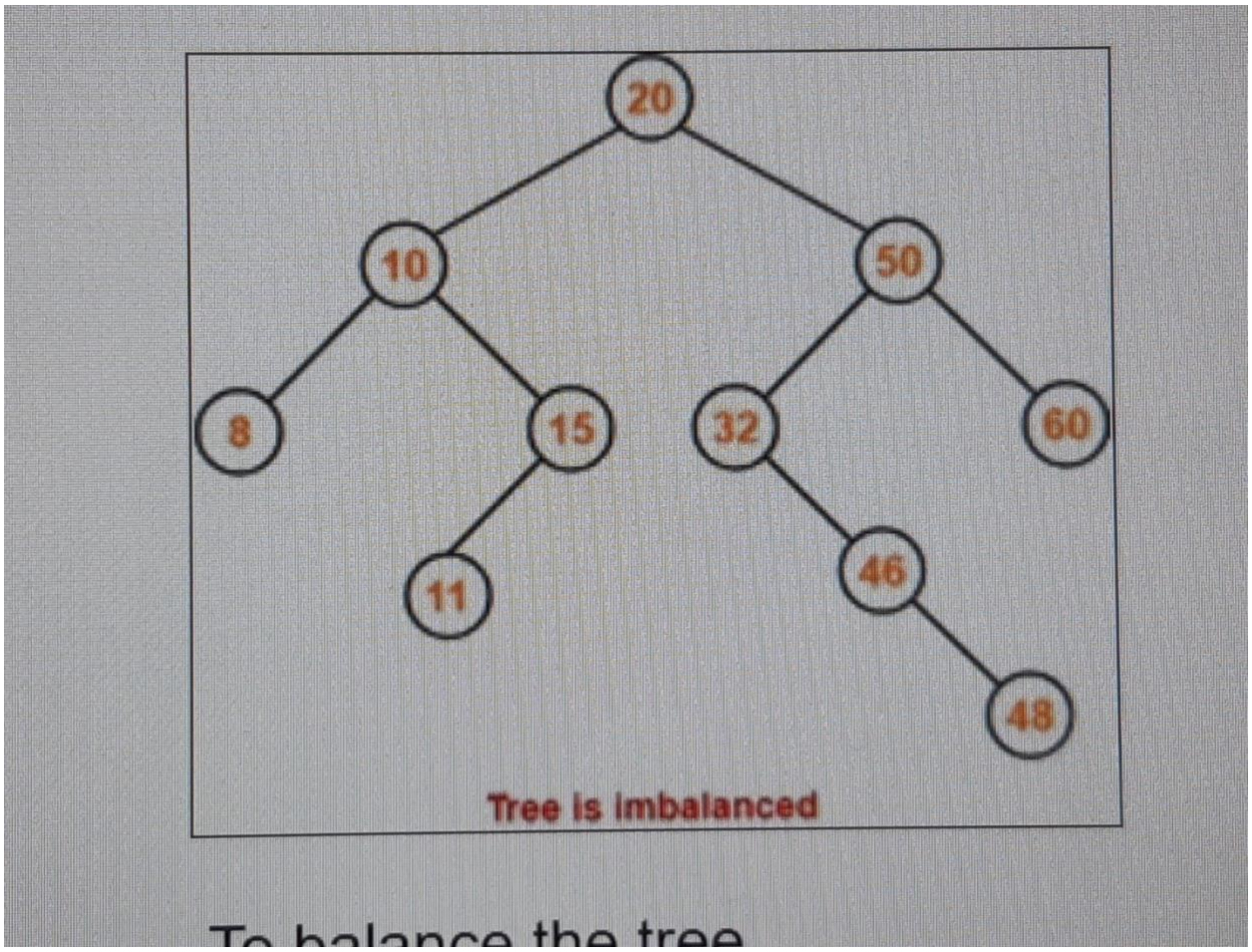
- As 48 > 20, so insert 48 in 20's right sub tree.

- As 48 < 50, so insert 48 in 50's left sub tree.
- As 48 > 32, so insert 48 in 32's right sub tree.
- As 48 > 46, so insert 48 in 46's right sub tree.
-

To balance the tree,

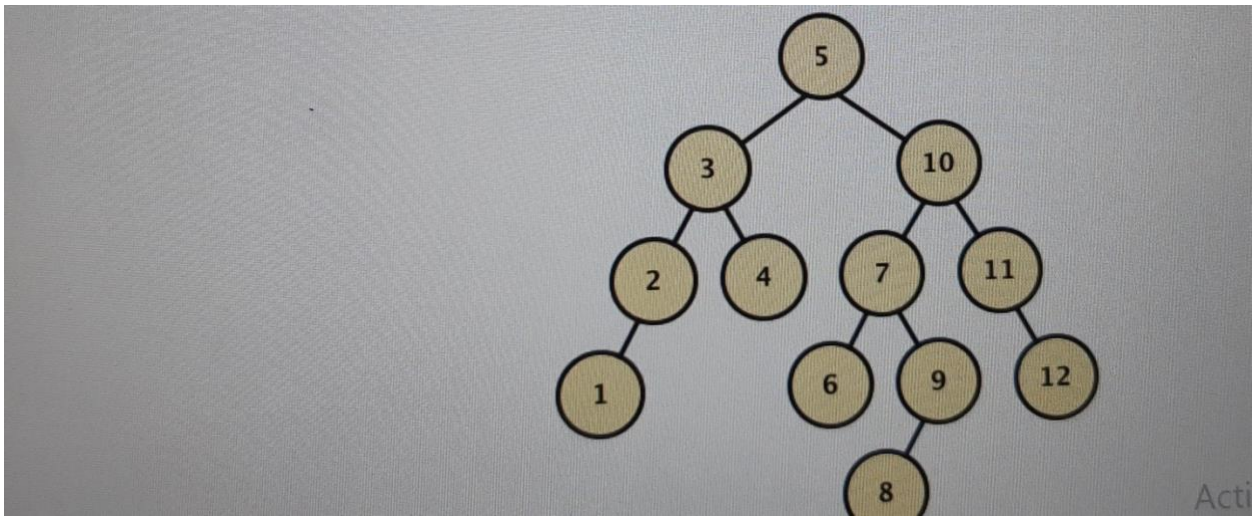
- Find the first imbalanced node on the path from the newly inserted node (node 48) to the root node.
- The first imbalanced node is node 32.
- Now, count three nodes from node 32 in the direction of leaf node.
- Then, use AVL tree rotation to balance the tree.

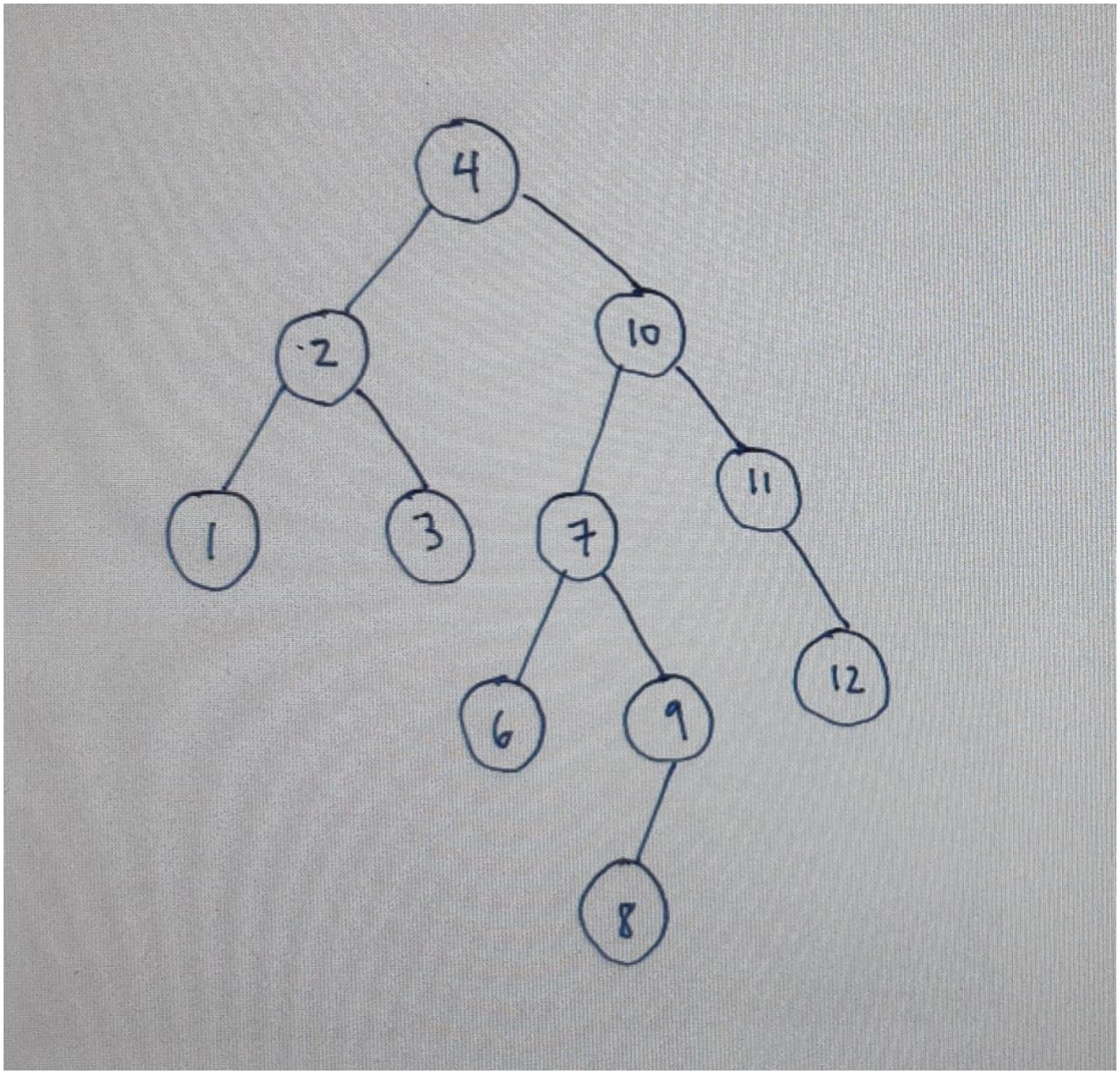
Following this, we have-



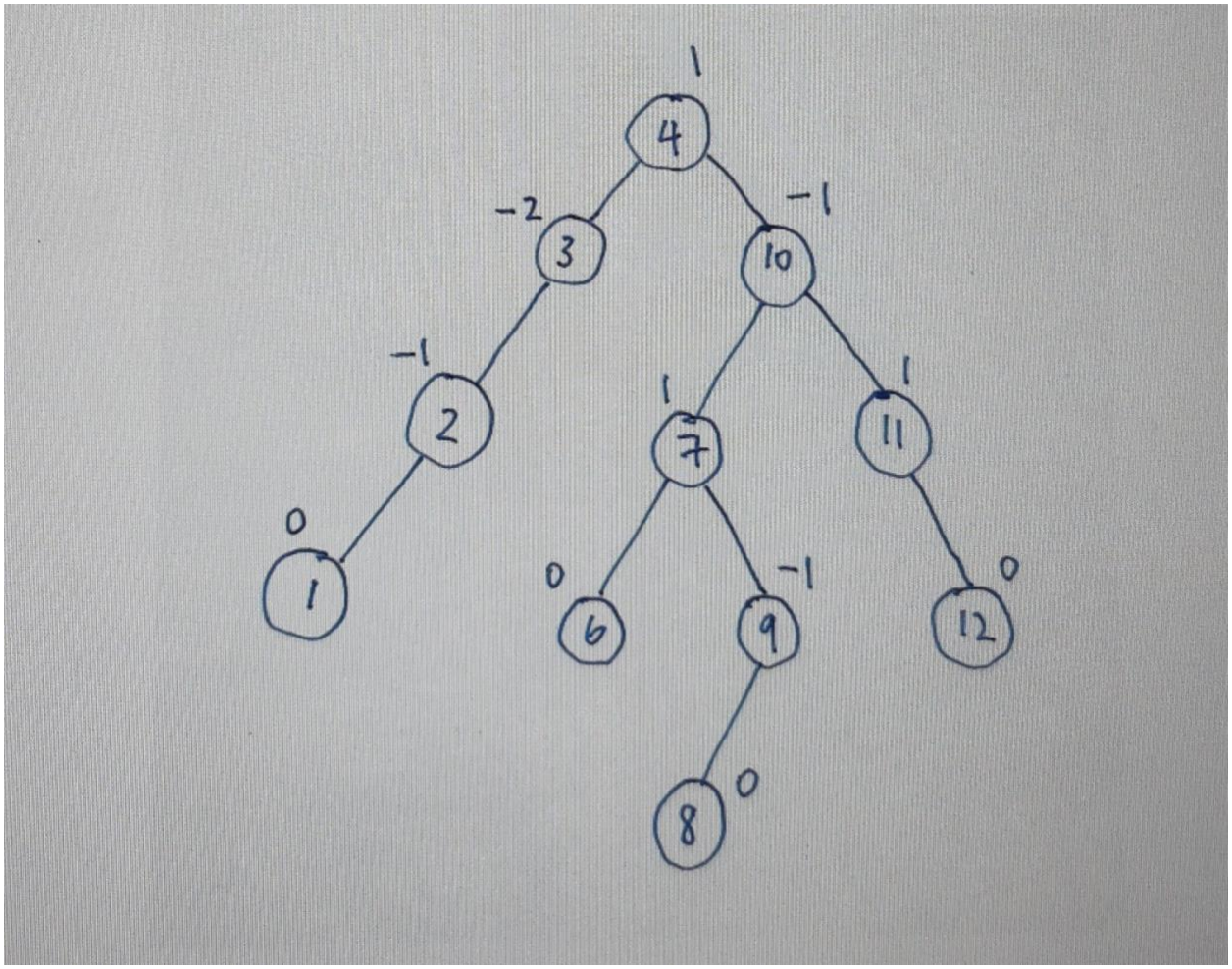
This is the final balanced AVL tree after inserting all the given elements

11. Draw the resulting BST after 5 is removed, but before any rebalancing takes place. Label each node in the resulting tree with its balance factor. Replace a node with both children using an appropriate value from the node's left child.





12. Now rebalance the tree that results from (a). Draw a new tree for each rotation that occurs when rebalancing the AVL Tree (you only need to draw one tree that results from an RL or LR rotation). You do not need to label these trees with balance factors.


1
4
-2
3
-1
10
-1
2
1
7
1
11
0
1
0
6
-1
9
0
12
0
8

We show the intermediate rotation from the RL rotation along with the final answer below: