Code - 2024

G. S. Mandal's

Maharashtra Institute of Technology, Aurangabad

(An Autonomous Institute)

END SEMESTER EXAMINATION

ECE

Second Year B.Tech (Branch) – Feb/Mar-2023

Model Answer (Scheme)

Course Code : ELE-203 **Course Name :** Data Struct & Algo.

Duration : 2 Hrs **Max. Marks : 50** **Date :**

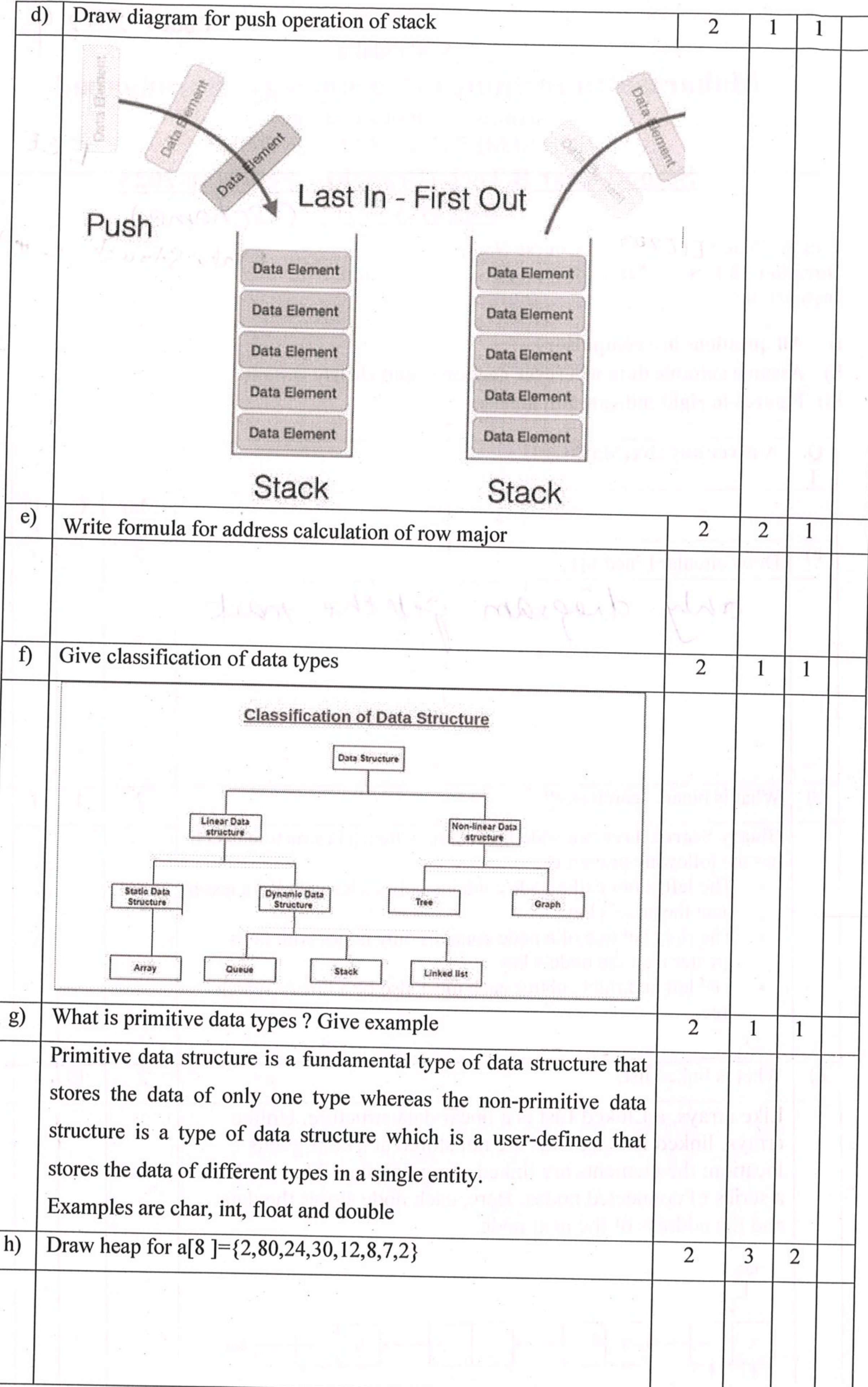
Instructions :

i) All questions are compulsory

ii) Assume suitable data wherever necessary and clearly state it

iii) Figures to right indicate full marks

Q. 1	Answer any five(Marks:10)				
		Marks	CO	BL	PI
a)	Draw circular Lined list	2	4	1	
	only diagram get the mark				
b)	What is binary search tree?	2	4	1	
	Binary Search Tree is a node-based binary tree data structure which has the following properties: • The left subtree of a node contains only nodes with keys lesser than the node's key. • The right subtree of a node contains only nodes with keys greater than the node's key. • The left and right subtree each must also be a binary search tree.				
c)	What is linked list?	2	1	1	
	Like arrays, a Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers. They include a series of connected nodes. Here, each node stores the data and the address of the next node. Head A → B → C → D → Null				



d)	Draw diagram for push operation of stack	2	1	1	
	(diagram)				
e)	Write formula for address calculation of row major	2	2	1	
f)	Give classification of data types	2	1	1	
	(diagram)				
g)	What is primitive data types ? Give example	2	1	1	
	Primitive data structure is a fundamental type of data structure that stores the data of only one type whereas the non-primitive data structure is a type of data structure which is a user-defined that stores the data of different types in a single entity. Examples are char, int, float and double				
h)	Draw heap for a[8]={2,80,24,30,12,8,7,2}	2	3	2	

Q. 2	Explain heap sort method by giving suitable example	8	3	2	

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

- Heap sort is an in-place algorithm.
- Its typical implementation is not stable, but can be made stable (See this)
- Typically 2-3 times slower than well-implemented QuickSort. The reason for slowness is a lack of locality of reference.

Consider the array: arr[] = {4, 10, 3, 5, 1}.

Build Complete Binary Tree: Build a complete binary tree from the array.

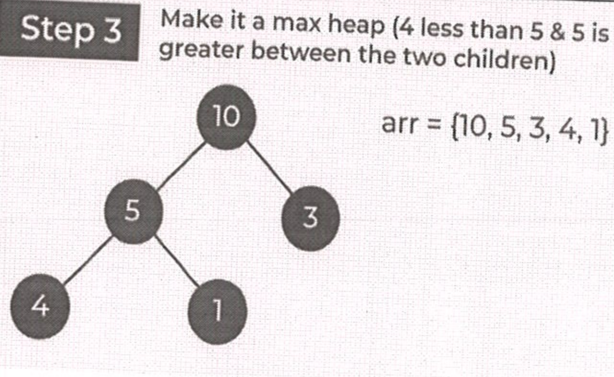
Build complete binary tree from the array

Transform into max heap: After that, the task is to construct a tree from that unsorted array and try to convert it into max heap.

- To transform a heap into a max-heap, the parent node should always be greater than or equal to the child nodes
    - Here, in this example, as the parent node **4** is smaller than the child node **10,** thus, swap them to build a max-heap.

Transform it into a max heap image widget

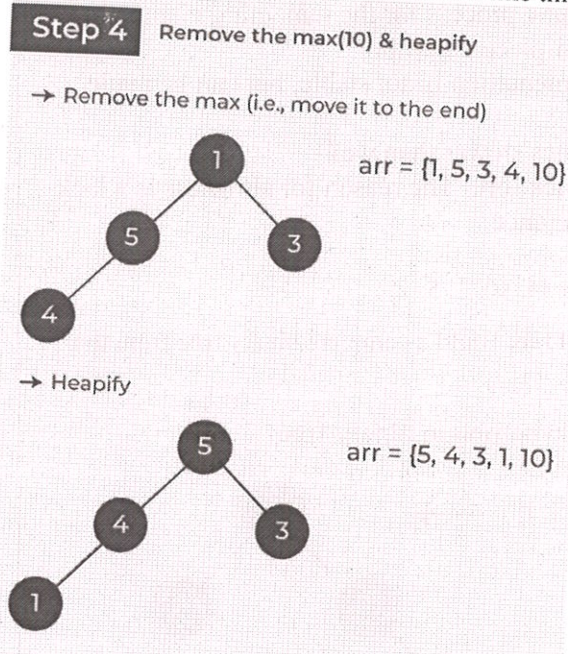
- Now, as seen, **4** as a parent is smaller than the child **5**, thus swap both of these again and the resulted heap and array should be like this:



Make the tree a max heap

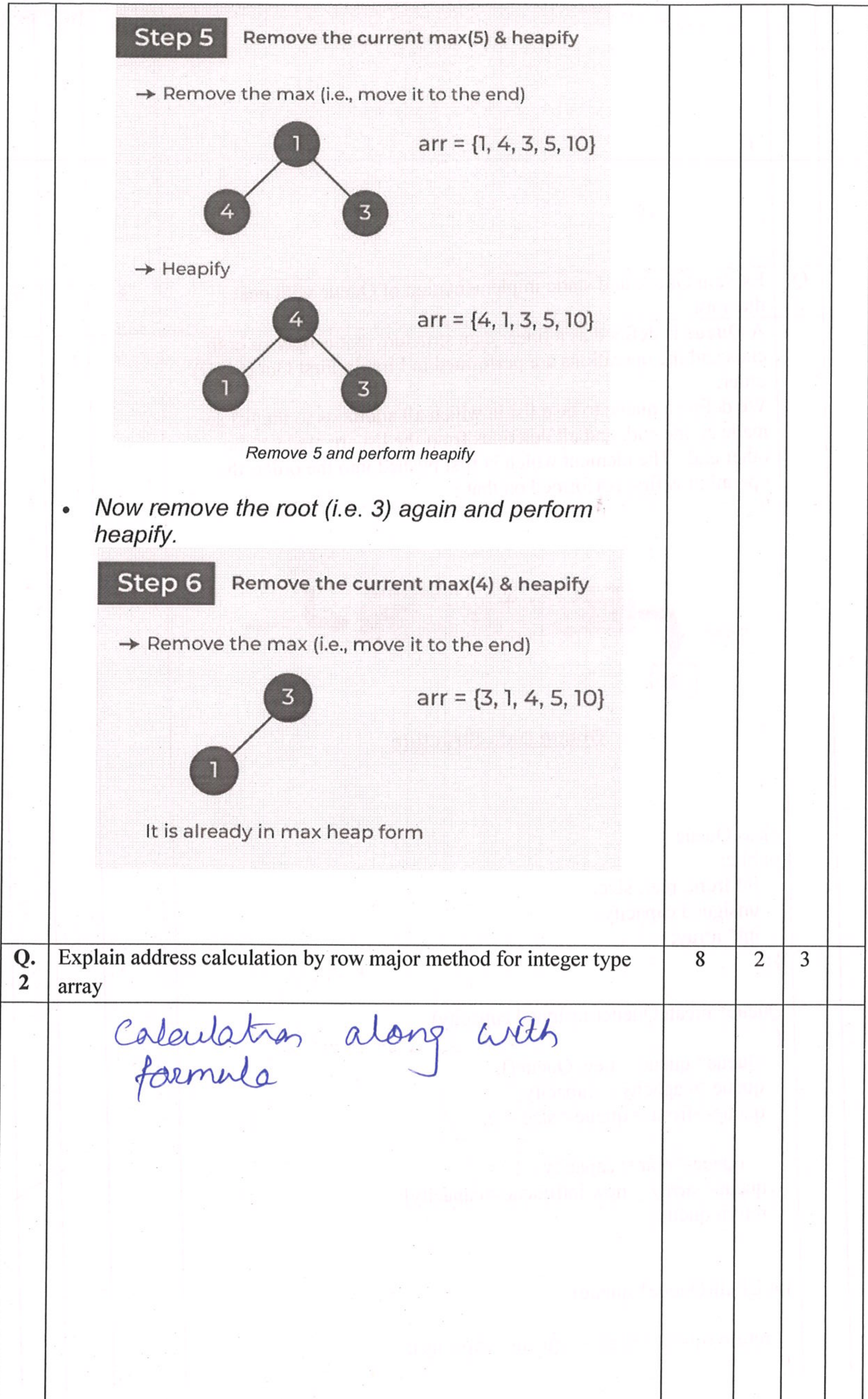
Perform heap sort: Remove the maximum element in each step (i.e., move it to the end position and remove that) and then consider the remaining elements and transform it into a max heap.

- Delete the root element **(10)** from the max heap. In order to delete this node, try to swap it with the last node, i.e. **(1).** After removing the root element, again heapify it to convert it into max heap.
    - Resulted heap and array should look like this:



Remove 10 and perform heapify

- Repeat the above steps and it will look like the following:



Remove 5 and perform heapify

- *Now remove the root (i.e. 3) again and perform heapify.*

Q. 2	Explain address calculation by row major method for integer type array	8	2	3
	Calculation along with formula			

Q. 3	Explain Queue and static implementation of Queue with neat diagram	8		3	

A **Queue** is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.

We define a queue to be a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end. The element which is first pushed into the order, the operation is first performed on that.

Queue Data Structure

```
class Queue {
public:
    int front, rear, size;
    unsigned capacity;
    int* array;
};

Queue* createQueue(unsigned capacity)
{
    Queue* queue = new Queue();
    queue->capacity = capacity;
    queue->front = queue->size = 0;

        queue->rear = capacity - 1;
    queue->array = new int[queue->capacity];
    return queue;
}

int isFull(Queue* queue)
{
    return (queue->size == queue->capacity);
}
```

```
int isEmpty(Queue* queue)
{
   return (queue->size == 0);
}

void enqueue(Queue* queue, int item)
{
   if (isFull(queue))
      return;
   queue->rear = (queue->rear + 1)
              % queue->capacity;
   queue->array[queue->rear] = item;
   queue->size = queue->size + 1;
   cout << item << " enqueued to queue\n";
}

int dequeue(Queue* queue)
{
   if (isEmpty(queue))
      return INT_MIN;
   int item = queue->array[queue->front];
   queue->front = (queue->front + 1)
              % queue->capacity;
   queue->size = queue->size - 1;
   return item;
}

int front(Queue* queue)
{
   if (isEmpty(queue))
      return INT_MIN;
   return queue->array[queue->front];
}

int rear(Queue* queue)
{
   if (isEmpty(queue))
      return INT_MIN;
   return queue->array[queue->rear];
}

int main()
{
   Queue* queue = createQueue(1000);

   enqueue(queue, 10);
   enqueue(queue, 20);
   enqueue(queue, 30);
   enqueue(queue, 40);

   cout << dequeue(queue)
      << " dequeued from queue\n";

   cout << "Front item is "
      << front(queue) << endl;
   cout << "Rear item is "
```

```
        << rear(queue) << endl;

    return 0;
}
```

Q. 3	Explain stack and implementation of stack with neat diagram	8	2	3

Stacks in Data Structures is a linear type of data structure that follows the LIFO (Last-In-First-Out) principle and allows insertion and deletion operations from one end of the stack data structure, that is top. Implementation of the stack can be done by contiguous memory which is an array, and non-contiguous memory which is a linked list. Stack plays a vital role in many applications.

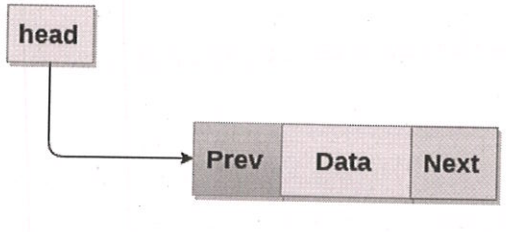
```
class Stack {
    int top;

    int a[MAX]; // Maximum size of Stack

    Stack() { top = -1; }
    bool push(int x);
    int pop();
        bool isEmpty();
};

bool Stack::push(int x)
{
    if (top >= (MAX - 1)) {
        printf( "Stack Overflow");
    }
    else {
        a[++top] = x;
        printf( " pushed into stack\n)";
    }
}

int Stack::pop()
{
    if (top < 0) {
        printf("Stack Underflow");
    }
    else {
        int x = a[top--];
        return x;
    }
}
int Stack::peek()
{
    if (top < 0) {
        prinf( "Stack is Empty)";
        return 0;
    }
    else {
        int x = a[top];
        return x;
    }
}

bool Stack::isEmpty()
{
```

```
    return (top < 0);
}

// Driver program to test above functions
int main()
{
    class Stack s;
    s.push(10);
    s.push(20);
    s.push(30);
    cout << s.pop() << " Popped from stack\n";

    //print top element of stack after popping
    cout << "Top element is : " << s.peek() << endl;

    //print all elements in stack :
    cout <<"Elements present in stack : ";
    while(!s.isEmpty())
    {
        // print top element in stack
        cout << s.peek() <<" ";
        s.pop();
    }
}
```

Q. 4	Sort the following array using merge sort method a[10]={27,15,78,45,32,10,8,4,90,6}	8	4	3	
	Step By step solve the problem each step carry marks				
Q. 4	Sort following array using insertion sort method a[6]={20,13,2,6,48,15,10}	8	4	3	

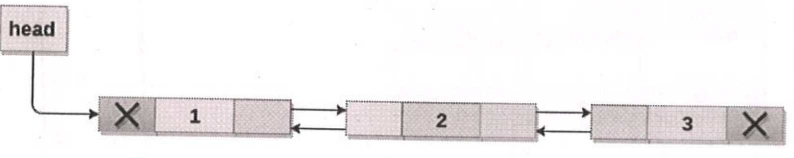
Q. 5	Explain binary search tree by giving suitable example	8	4	3	
	A binary Search Tree is a special tree in which some order is followed. Every parent node has at most two children in which the left children have a lesser value while the right children have a higher value than their parent. This rule is applied to all left and right subtrees. Binary Search Tree Not a Binary Search Tree because, 21 > 20 insert the new node, we have to follow the given steps: 1. Find the node where the new node has to be inserted and store the visited node in the temp variable.				

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



Node

A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



Doubly Linked List

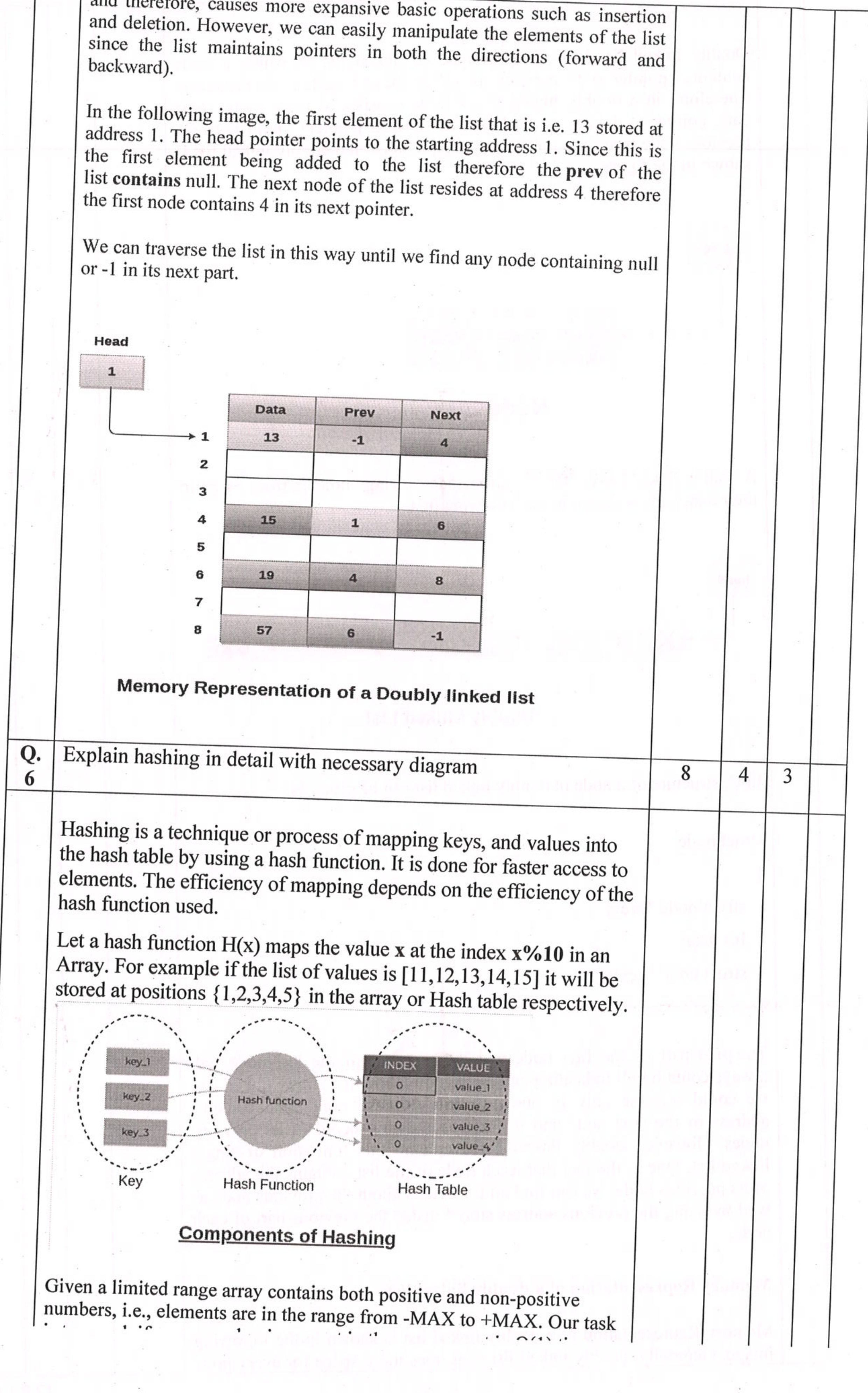
In C, structure of a node in doubly linked list can be given as :

```
struct node
{
  struct node *prev;
  int data;
  struct node *next;
}
```

The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

Memory Representation of a doubly linked list

Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node



and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).

In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the **prev** of the list **contains** null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

We can traverse the list in this way until we find any node containing null or -1 in its next part.

Head: 1

	Data	Prev	Next
1	13	-1	4
2			
3			
4	15	1	6
5			
6	19	4	8
7			
8	57	6	-1

Memory Representation of a Doubly linked list

Q. 6	Explain hashing in detail with necessary diagram	8	4	3

Hashing is a technique or process of mapping keys, and values into the hash table by using a hash function. It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used.

Let a hash function H(x) maps the value **x** at the index **x%10** in an Array. For example if the list of values is [11,12,13,14,15] it will be stored at positions {1,2,3,4,5} in the array or Hash table respectively.

Components of Hashing

Given a limited range array contains both positive and non-positive numbers, i.e., elements are in the range from -MAX to +MAX. Our task

Since the range is limited, we can use index mapping (or trivial hashing). We use values as the index in a big array. Therefore we can search and insert elements in O(1) time.

T:

Key = 21 → 21 → data for Key = 21

20
21
22

Avoid branching[edit]

Roger Sayle gives an example[2] of eliminating a multiway branch caused by a switch statement:

```
inline bool HasOnly30Days(int m)
{
        switch (m) {
        case 4:  // April
        case 6:  // June
        case 9:  // September
        case 11: // November
                return true;
        default:
                return false;
        }
}
```

Which can be replaced with a table lookup:

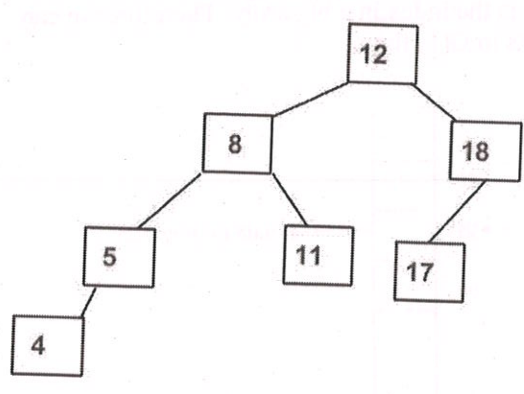
```
inline bool HasOnly30Days(int m)
{
        static const bool T[] = { 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0 };
        return T[m-1];
}
```

Q. 6	Explain AVL-tree with neat diagram	8	4	3	

AVL Tree:

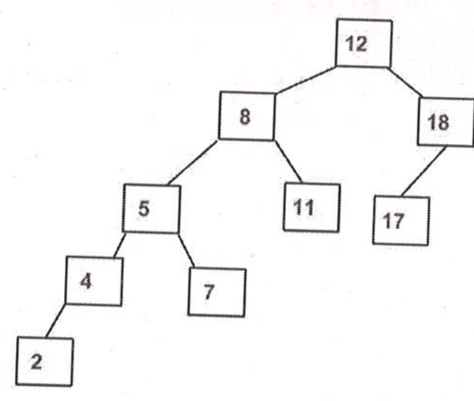
AVL tree is a self-balancing Binary Search Tree (**BST**) where the difference between heights of left and right subtrees cannot be more than **one** for all nodes.

Example of AVL Tree:



The above tree is AVL because the differences between the heights of left and right subtrees for every node are less than or equal to 1.

Example of a Tree that is NOT an AVL Tree:



The above tree is not AVL because the differences between the heights of the left and right subtrees for 8 and 12 are greater than 1.

Why AVL Trees?

*Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take **O(h)** time where **h** is the height of the BST. The cost of these operations may become **O(n)** for a **skewed Binary tree**. If we make sure that the height of the tree remains **O(log(n))** after every insertion and deletion, then we can guarantee an upper bound of **O(log(n))** for all these operations. The height of an AVL tree is always **O(log(n))** where **n** is the number of nodes in the tree.*

Insertion in AVL Tree:

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing.

Following are two basic operations that can be performed to balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

- Left Rotation
- Right Rotation

T1, T2 and T3 are subtrees of the tree, rooted with y (on the left side) or x (on the right side)

y x

```
  /\    Right Rotation        / \
 x  T3  ------->        T1  y
 /\     <-------            /\
T1  T2   Left Rotation        T2  T3
```

Keys in both of the above trees follow the following order

keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)

Submission count: 12.1K

Steps to follow for insertion:

Let the newly inserted node be **w**

- Perform standard **BST** insert for **w.**
- Starting from **w**, travel up and find the first **unbalanced node**.
  Let **z** be the first unbalanced node, **y** be the **child** of **z** that comes on the path from **w** to **z** and **x** be the **grandchild** of **z** that comes on the path from **w** to **z.**
- Re-balance the tree by performing appropriate rotations on the subtree rooted with **z.** There can be 4 possible cases that need to be handled as **x, y** and **z** can be arranged in 4 ways.
- Following are the possible 4 arrangements:
    - y is the left child of z and x is the left child of y (Left Left Case)
    - y is the left child of z and x is the right child of y (Left Right Case)
    - y is the right child of z and x is the right child of y (Right Right Case)
    - y is the right child of z and x is the left child of y (Right Left Case)

1. Left Left Case

T1, T2, T3 and T4 are subtrees.

```
     z                          y
    /\                         / \
   y  T4    Right Rotate (z)       x    z
  /\        --------->        / \  / \
 x  T3                      T1  T2  T3  T4
 /\
T1  T2
```

2. Left Right Case

```
   z                    z                    x
  /\                   / \                  / \
 y  T4  Left Rotate (y)      x   T4  Right Rotate(z)   y     z
 /\     --------->    / \     -------->  /\   /\
T1  x                y   T3              T1  T2 T3  T4
   /\               /\
  T2  T3           T1  T2
```

3. Right Right Case

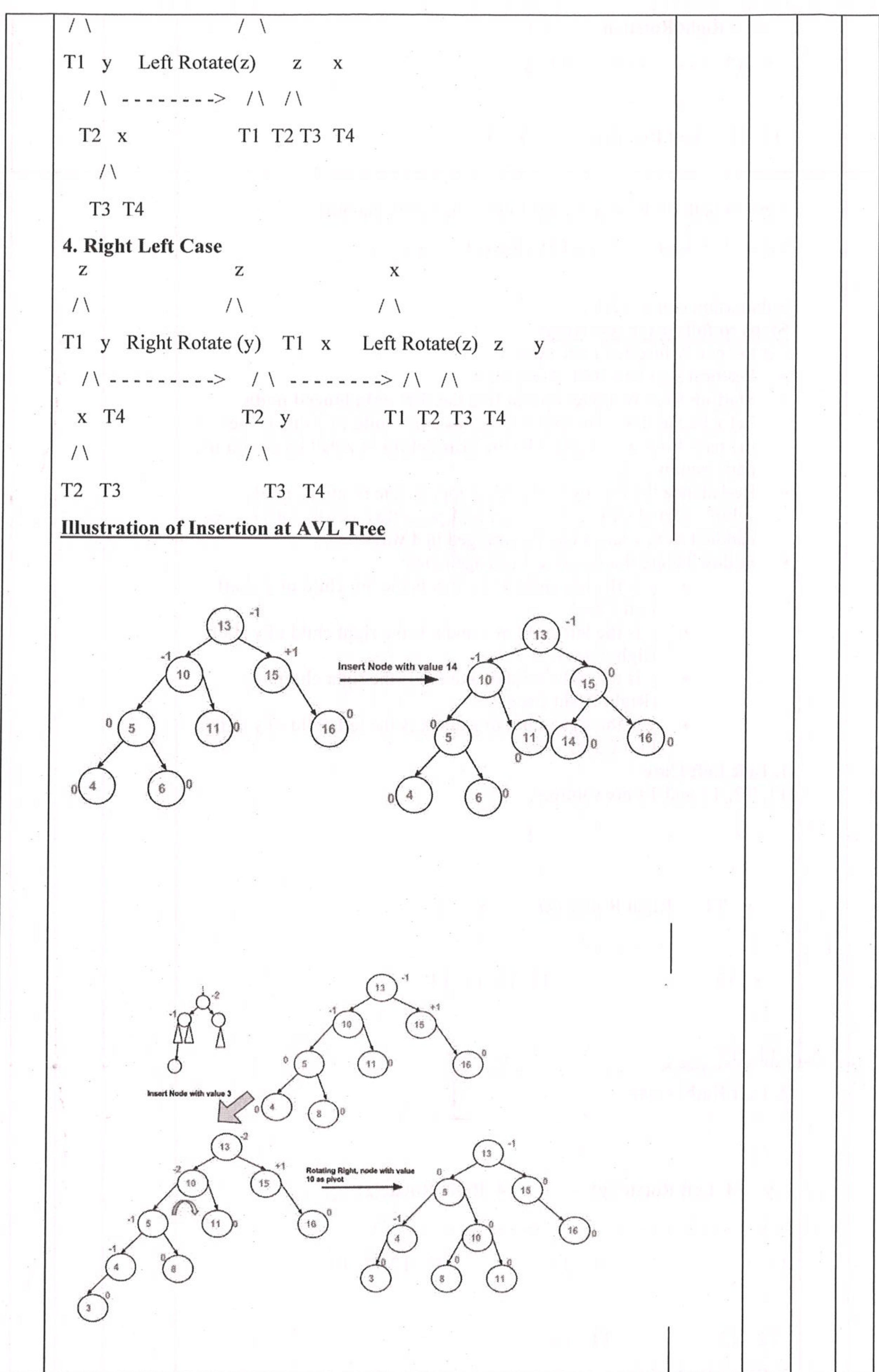
```
 z                    y
```

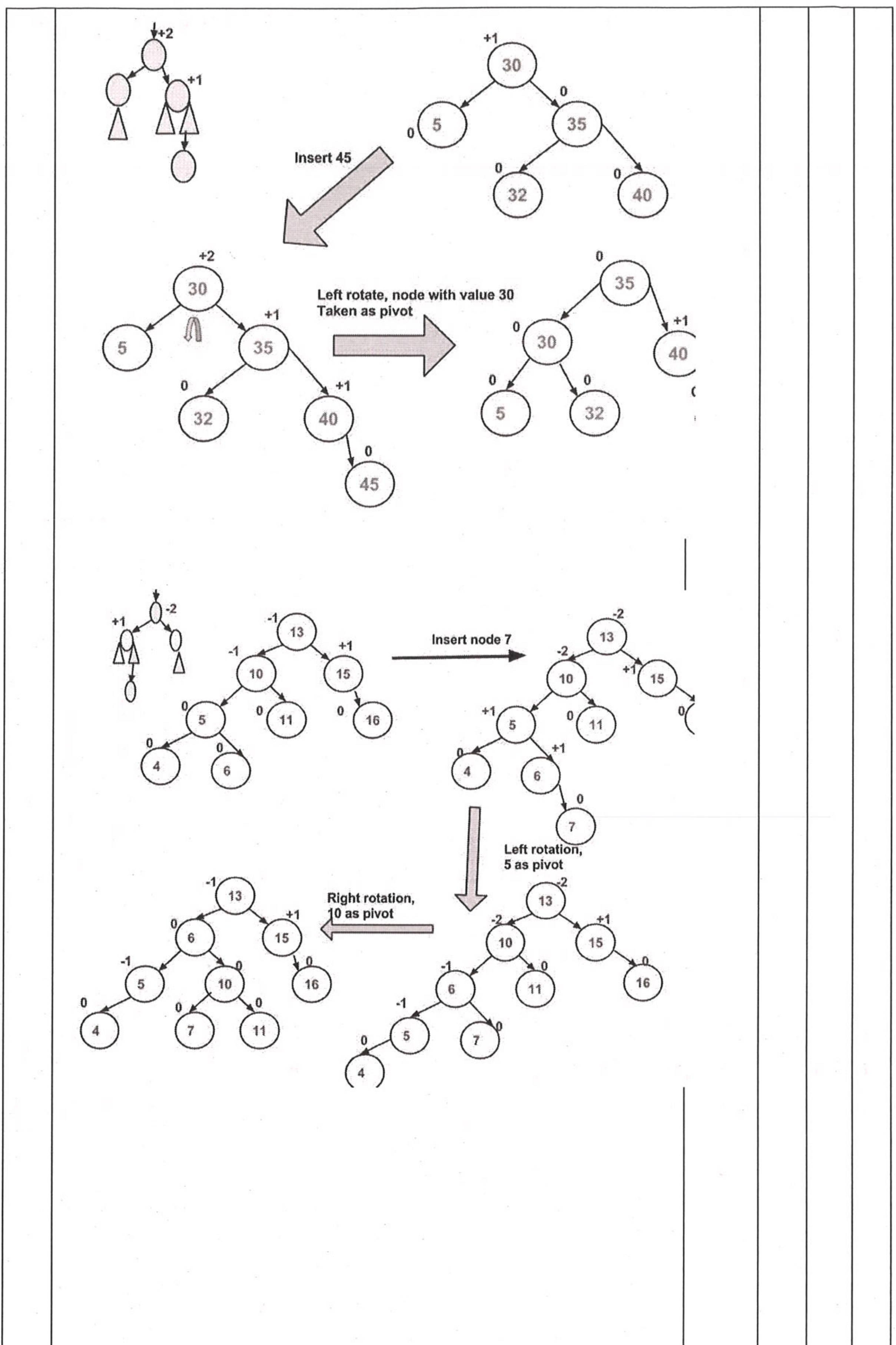
```
      / \                    / \
T1   y    Left Rotate(z)      z     x
      / \  -------->     /\   /\
   T2   x                T1  T2 T3  T4
         /\
       T3  T4
```

4. Right Left Case

```
   z                      z                       x
  /\                     /\                      / \
T1   y  Right Rotate (y)  T1   x    Left Rotate(z)  z     y
   /\ --------->       / \  -------->  /\   /\
  x  T4                T2   y          T1  T2  T3  T4
 /\                        / \
T2  T3                     T3  T4
```

<u>Illustration of Insertion at AVL Tree</u>





Note:- All course outcomes shall be addressed.