CBCS SCHEME

BCS304

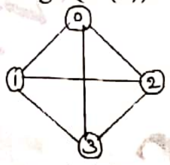
USN | 2 | V | D | 2 | 2 | C | I | 0 | 0 | 5

Third Semester B.E./B.Tech. Degree Examination, Dec.2023/Jan.2024
Data Structures and Applications

Time: 3 hrs. Max. Marks: 100

Note: 1. Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.

		Module – 1	M	L	C
Q.1	a.	Define Data Structures. Explain with neat block schematic different type of data structures with examples. What are the primitive operations that can be performed?	10	L2	CO1
	b.	Differentiate between structures and unions shown examples for both.	5	L1	CO1
	c.	What do you mean by pattern matching? Outline knuth, Morris, Pratt pattern matching algorithm.	5	L2	CO1
		OR			
Q.2	a.	Define stack. Give the implementation of Push ( ), POP ( ) and display ( ) functions by considering its empty and full conditions.	7	L2	CO1
	b.	Write an algorithm to evaluate a postfix expression and apply the same for the given postfix expression 6, 2, /, 3, -, 4, 2, *, +	7	L3	CO1
	c.	Write the Postfix form of the following using stack : (i) A*(B*C+D*E) + F (ii) (a + (b*c) / (d-e))	6	L3	CO1
		Module – 2			
Q.3	a.	What are the disadvantages of ordinary queue? Discuss the implementation of circular queue.	8	L2	CO2
	b.	Write a note on multiple stacks and priority queue.	6	L2	CO2
	c.	Define Queue. Discuss how to represent queue using dynamic arrays.	6	L2	CO2
		OR			
Q.4	a.	What is a linked list? Explain the different types of linked lists with neat diagram.	4	L2	CO2
	b.	Give the structure definition for singly linked list (SLL). Write a C function to, (i) Insert on element at the end of SLL. (ii) Delete a node at the beginning of SLL.	8	L3	CO2
	c.	Write a C-function to add two polynomials show the linked list representation of below two polynomials $p(x) = 3x^{14} + 2x^{8} + 1$ $q(x) = 8x^{14} - 3x^{10} + 10x^{6}$	8	L3	CO2
		Module – 3			
Q.5	a.	Write a C-function for the following operations on Doubly Linked List (DLL): (i) addition of a node. (ii) concatenation of two DLL.	8	L3	CO3
	b.	Write C functions for the following operations on circular linked list : (i) Inserting at the front of a list. (ii) Finding the length of a circular list.	8	L3	CO3

	c.	For the given sparse matrix, give the diagrammatic linked representation. $A = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$.	4	L3	CO3
		OR			
Q.6	a.	Discuss how binary tree are represented using, (i) Array (ii) Linked list	6	L2	CO3
	b.	Discuss inorder, preorder, postorder and level order traversal with suitable recursive function for each.	8	L2	CO3
	c.	Define Threaded Binary Tree. Discuss In-Threaded binary Tree.	6	L2	CO3
		Module – 4			
Q.7	a.	Write a function to perform the following operations on Binary Search Tree (BST) : (i) Inserting an element into BST. (ii) Recursive search of a BST.	8	L3	CO4
	b.	Discuss selection Trees with an example.	8	L2	CO4
	c.	Explain Transforming a ~~first~~ forest into a binary tree with an example.	4	L2	CO4
		OR			
Q.8	a.	Define graph. Show the adjacency matrix and adjacency list representation of the graph given below (Refer Fig. Q8 (a)).  Fig. Q8 (a)	6	L3	CO4
	b.	Define the following Terminologies with examples, (i) Digraph (ii) Weighted graph (iii) Self loop (iv) Parallel edges	8	L1	CO4
	c.	Explain in detail elementary graph operations.	6	L2	CO4
		Module – 5			
Q.9	a.	What is collision? What are the methods to resolve collision? Explain linear probing with an example.	7	L2	CO5
	b.	Explain in detail, about static and dynamic hashing.	6	L2	CO5
	c.	Discuss Leftist Trees with an example.	7	L2	CO5
		OR			
Q.10	a.	Explain different types of HASH function with example.	6	L2	CO5
	b.	Discuss AVL tree with an example. Write a function for insertion into an AVL Tree.	6	L3	CO5
	c.	Define Red-black Tree, Splay tree. Discuss the method to insert an element into Red-Black tree.	8	L2	CO5

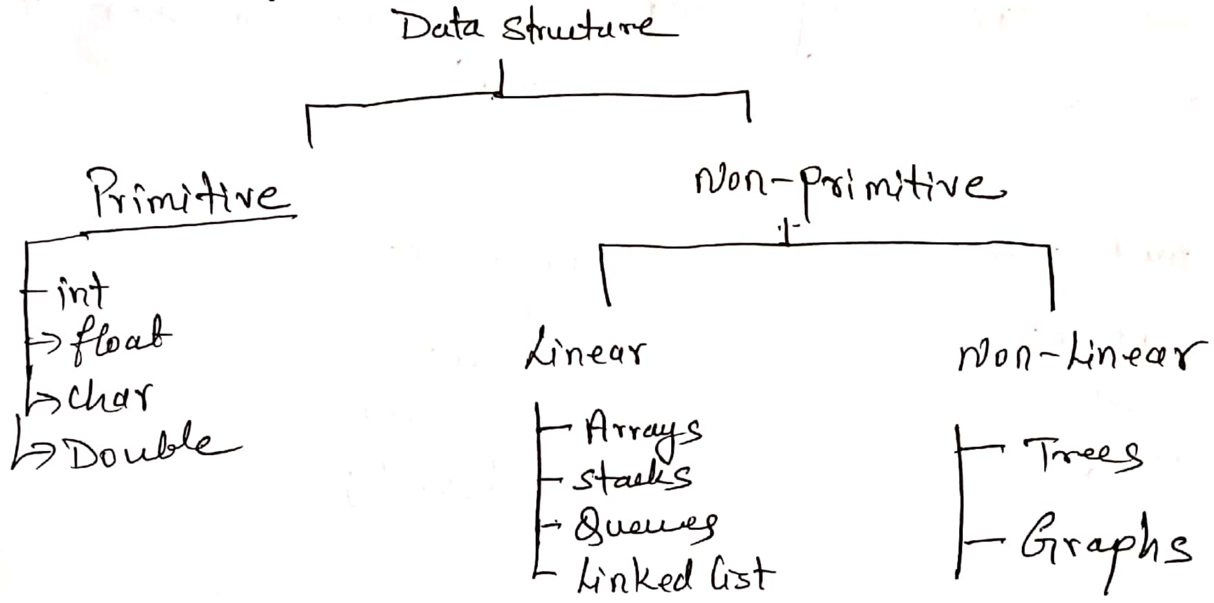
* * * * *

Semester : III

Subject : Data structures and Applications

Subject code : BCS304.

1a) Data Structures :- It can be defined as a method of storing & organizing the data items in the computer's memory.



Primitive :- The data structures, that are directly operated upon by machine level, instructions, eg:- int, float, char,

Non Primitive :- The data str can't be manipulated directly by machine level instructions are called Non-primitive eg:- arrays, stacks, queues, Linked List. trees, Graphs

operations are. inserting, deleting, Merging, Searching. Sorting (either in Ascending order or Descending order).

1b. Structure | Union.

Structure	Union.
Struct Keyword	Union Keyword.
Unique Memory Location,	Shared Memory Location.
Changing the value of one data member will not affect other data member.	Changing the value of one data it will affect other data members
we can retrieve any member at a time.	Only one member at a time

eg:-

```
struct dsc
{
    int a;
    float b;
} cse;
```

```
union dsc
{
    int a;
    float b;
} aiml;
```

1c) It is the fundamental problem in computer science where the goal is to find all occurences of a given pattern whithin a larger text or string.

```
Void stringmatch()
{  while(str[c] != '\0')
   {   if(str [m] == pat [i])
       {    i++;  m++;
          if (pat [i] != '\0')
          {   flag=1;
              for (K=0; rep[K] != '\0'; K++, j++)
              {
                  ans[j] = rep[K];
              }
              c=m;
   }   }
```

1a) Stack :- It is a linear data structure in which items are inserted and deleted at only one end called top of the stack.

```
void push()
{
    if (top >= n-1)
     { printf("Stack is overflow")
     }
    else
     {
       printf("Enter an item")
       sf("%d", &item)
       top++;
       stk[top] = item;
     }
  }

void pop()
{
    if (top == -1)
    { printf("Stack is underflow")
    }
    else
    {
       printf(" the popped element is %d", stk[top])
       top--;
    }
  }

void display()
{
    if (top == -1)
      pf(" stack is empty")
    else
     { for(i= top; i>=0; i--)
         pf("%d", stk[i])
  }  }
```

2b) Evaluate a post fix expression.

→ scan the postfix expression from left to right

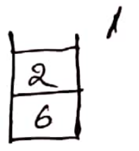
→ If scanned symbol is an operand then push it onto the stackk

→ if the symbol is an operator then pop out two symbols from the stack and assign to.

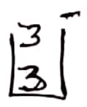
<B> <operator> <A>

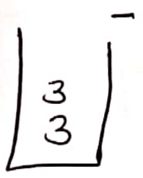
→ Repeat · step 2 and 3 till end of the expression

6 2 / 3 - 4 2 * +

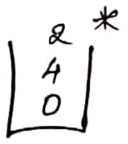


⇒ 6/2 = 3 push onto the stack, then Scan next symbol. 3 again push to stack

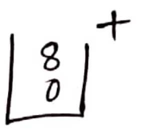




3 - 3 = 0· push on to the stack.



popout, 2 * 4 = 8. push on to the stack



8 + 0 = 8 push ~~on~~ on to the stack

then finally popout from the stack

answer 8//

2c) Infix to Postfix :-

(i)

	Stack	Res
A	—	A
*	*	A
(	*(	A
B	*(	AB
*	*(*	AB
C	*(*	ABC
+	*(+	ABC*
D	*(+	ABC*D
*	*(+*	ABC*D
E	*(+*	ABC*DE
)	*$\boxed{(+*)}$	ABC*DE*+
+	*	ABC*DE*+*
F	+	$\boxed{ABC*DE*+*F+}$

(ii)

(	(	—
a	(	a
+	(+	a
(	(+(	a
b	(+(	ab
*	(+(*	ab
c	(+(*	abc
)	(+$\boxed{(*)}$	abc*
/	(+/	abc*
(	(+/(	abc*
d	(+/(	abc*d
-	(+/(-	abc*d
e	(+/(-	abc*de
)	(+/$\boxed{(-)}$	abc*de-
)	$\boxed{(+/)}$	abc+de-/+

$\boxed{abc+de-/+}$

3a.) Dis advantages of ordinary Queue

→ Insertion & deletion of elments are time consumming.

→ When Queue is full we can't insert new element in ordinary Queue.

→ To overcome this problem we should implement Circular Queue.

```
Void enq()
{
    if (f == (r+1)%size)
    {
        pf("CQ is overflow")
    }
    else
    {   pf(" enter data")
        sf("%d", & data)
        if (f==-1)
            f=0;
        r=(r+1)%size;
        cq[r]=data;
    } }

Void deq()
{   if (f==-1)
    pf("CQ is underflow");
    else
    { pf(" Deleted data is %d", cq[f]);
      if (f==r)
         f=r=-1;
      else                       f=(f+1)%size;
                                 }
            →              }
```

Scanned with CamScanner

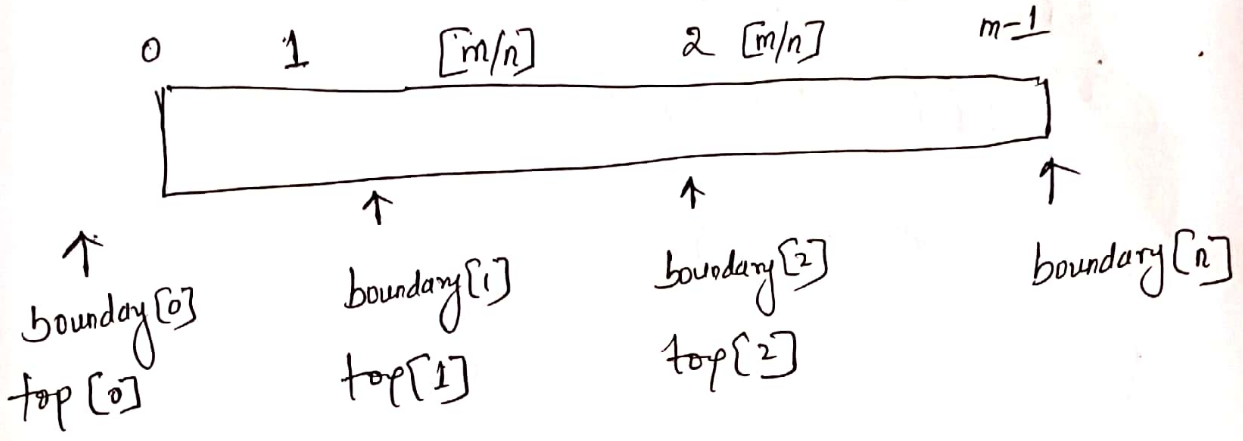
```
void display()
{
    if(f==-1)
    {
     pf("cq is empty")
    }
  else
  {
    for(i=f; i!=r; i=(i+1)%size)
    {    pf("%d", cq[i])
    }
    pf("%d", cq[i])
    }
  }
```

3b) Multiple stacks and Priority Queue :-

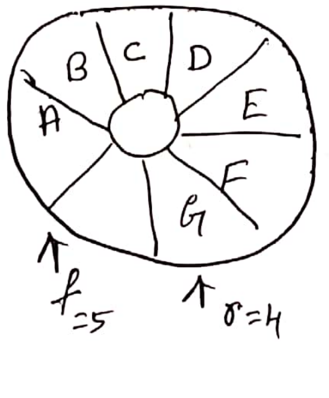
More than one stacks can be implemented on a single one dimensional array.
Size of each array can be equal or different.
Number of arrays can be fixed or varying.
Simple implementation consists of two arrays of size of each array can not be predicted.
Boundary condition.
$B(i) = T(i)$ iff the ith stack is empty

```
#define msize 100
#define maxstack 10
  element memory[msize];
  int top[maxstack];
  int boundary[maxstack];
  int n;
```

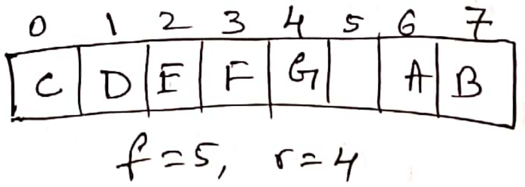
```
top[0] = boundary[0] = -1
for (i = 1; i<n; i++)
  top[i] = boundary[i] = (memory size/n) * i;
boundary[n] = memorysize -1;
```



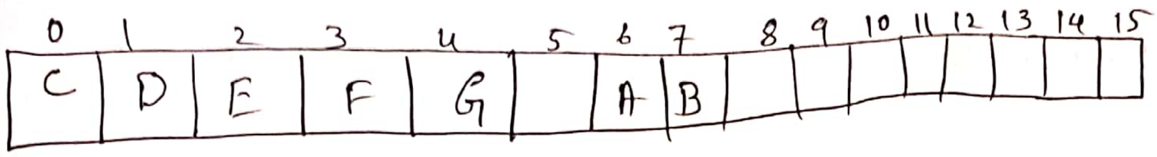
3c) Queue :- It is a linear datastructure in which items are inserted at one end called "rear" & items are deleted at another end called 'front' end. also known as FIFO.



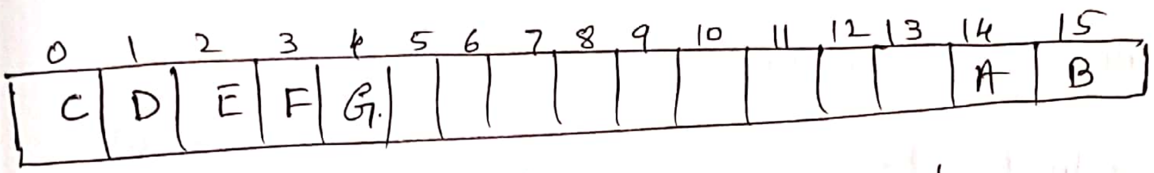
Circular Queue.



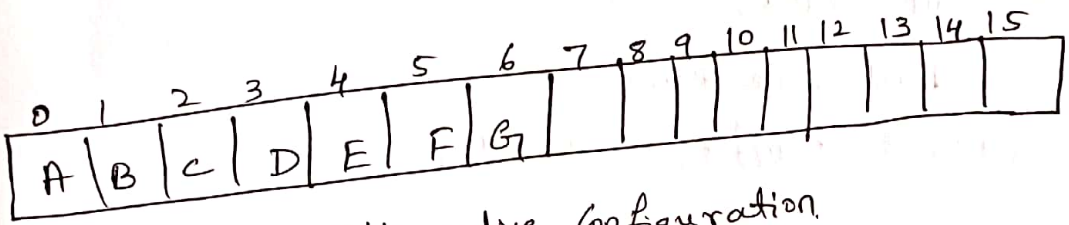
Flattened view of CQ.



After array doubling.



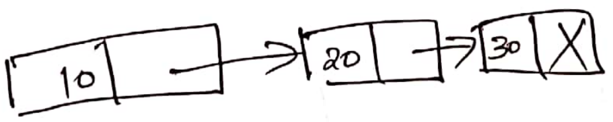
After shifting right segment.



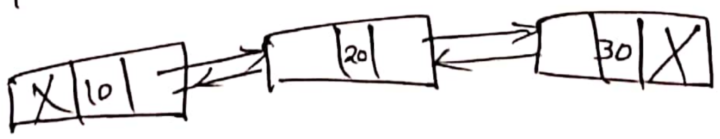
Alternative Configuration.

4a) Linked list is a linear data structure where elements are stored in sequence & each element points to the next node it contains data & link fields

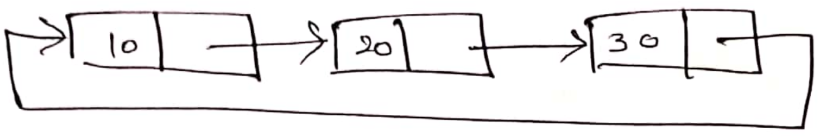
1) Singly LL :- each node points to the next node in the sequence and last node points to NULL.



2) Doubly linked list :- each node has two pointers one that points to right and another is to left link.



Circular LL :- Here last node is connected back to the first node. It can be either SLL or DLL.



4b) Structure definition :-

```
struct node
{
        int data;
        struct node *link;
} *head, *tmp, *ptr = NULL;
```

* Insert an element at the end of SLL

```
Void insertEnd()
{
  ptr = (*list) malloc(sizeof(list))
  pf(" Enter data")
  sf("%d", &data)
  ptr->data = num;
  ptr->link = NULL;
  if (head == NULL)
     head = ptr;
   else
   { tmp = head;
     while (tmp->link != NULL)
        tmp = tmp->link;
        tmp->link = ptr;
   }
}
```

Scanned with CamScanner

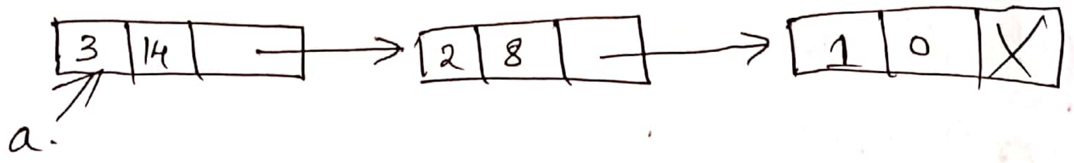
* Delete beginning of SLL.

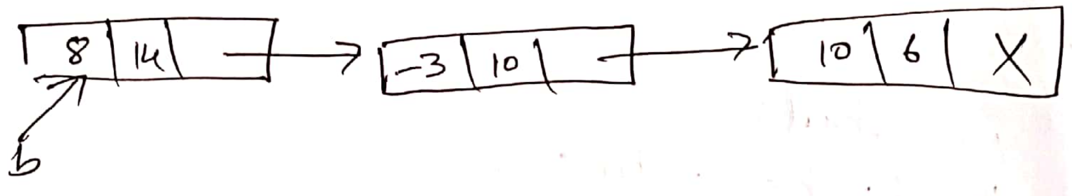
```
void delbeg()
{
    if(head == NULL)
    { pf(" SLL is empty")
    }
    else
    { ptr = ~~head~~;
      ~~s~~head = head → link;
      pf("deleted is %d", ptr → data);
      free(ptr);
    }
}
```

4c)
```
    poly-ptr padd (poly-ptr a, poly-ptr b)
    {
      poly-ptr front, rear, tmp;
      int sum;
      while (a && b)
      {
      switch (Compare (a → expon, b → expon))
      {   case -1 : attach (b → coef, b → exp, &rear);
                    b = b → link; break;

          case 0 :  sum = a → coef + b → coef;
                    if (sum) attach (sum, a → exp, &rear)
                    a = a → link;  b = b → link; break;

          case 1 :  attach (a → coef, a → exp. &rear)
                    a = a → link;
      }
    } }
```

$P(x) = 3x^{14} + 2x^{8} + 1$

$q(x) = 8x^{14} - 3x^{10} + 10x^{6}.$





5a) <u>ADDITION</u> of a node

i)
```
Void add (node ** headref, int data)
{
    node * new = createNode (data)
    if (* headref == NULL)
      {
         * headref = newnode;
         return;
      }
    node * current = *headref;
    while (current -> next != NULL)
    {
        current = current -> next;
     }
     current -> next = newnode;
      newnode -> prev = current;
    }
```

ii)
```
Void Concatinate (shut node * first, shut node * second)
  {     shut node * p = first;
      while (p -> next != NULL)
        {
            p = p -> next;
          }
        p -> next = second;
          second = NULL;
     } }
```

Circular linked list

i) Insert front

```
Void insrtfront (**headref, int data)
{
    node * newnode = crtnode(data);
    If (*headref == NULL)
    {   *headref = newnode;
        newnode -> next = newnode;
    }
    else
    {
        node * last = *headref;
        while (last -> next != *headref)
        {   last = last -> next;
        }
        newnode -> next = *headref;
        last -> next = newnode;
        *headref = newnode;
    }
}
```

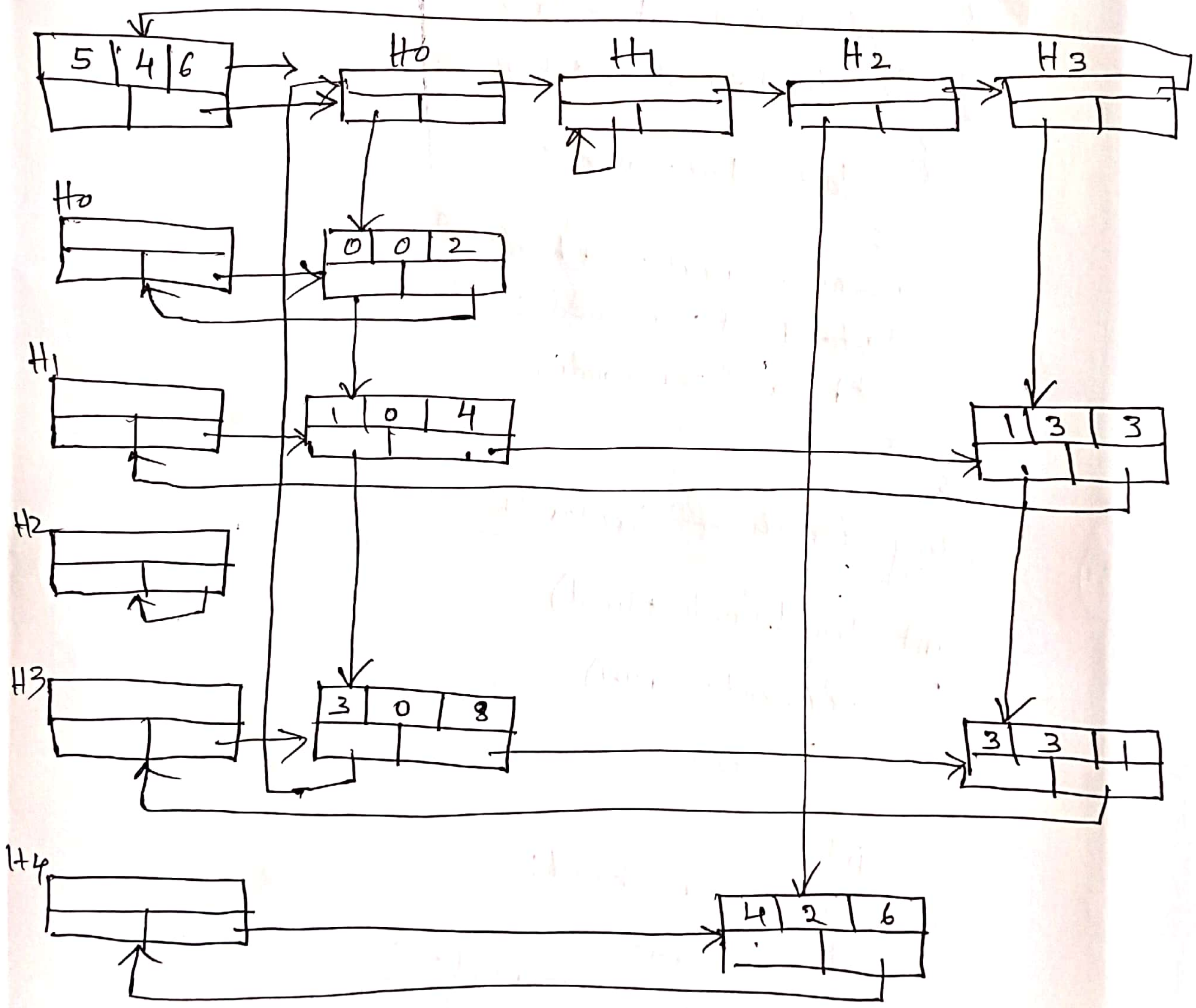
ii) Finding length of Circular LL;

```
int length (node *head)
{   if (head == NULL)
      { return 0;
      }
    int len = 0;
    node * cur = head;
    do{ len ++;
        cur = cur -> next;
    }while (cur != head);
    return length;
}
```

5c) Triplet form :-

a)

row	col	value
5	4	6
0	0	2
1	0	4
1	3	3
3	0	8
3	3	1
4	2	6



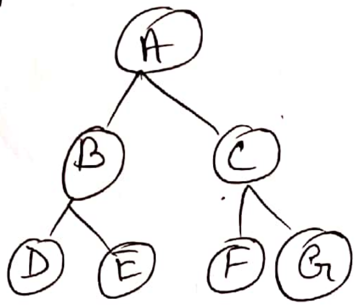
Scanned with CamScanner

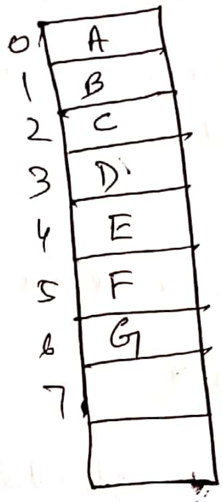
6a) Binary Tree Representation

1) Array :- Also called sequential Representation.
The nodes are numbered from 0 to n & one dimensional array can be used to store the nodes.

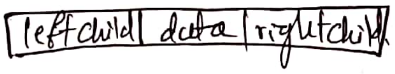
root node → a[0]
Left child → 2i + 1
right child → 2i + 2.

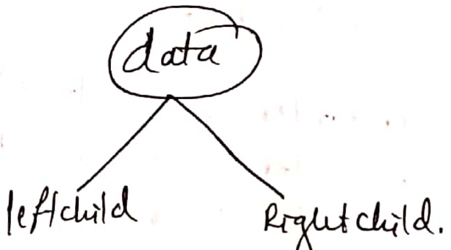
eg:-



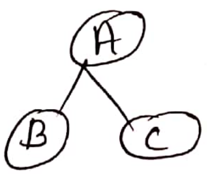


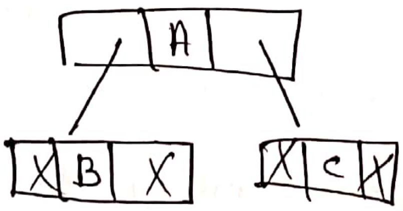
ii) Linked list :- Each node has 3 fields.
leftchild → which contains the address of left subtree
Data → which contains the actual information.
Rightchild → which contains the address of right subtree.





eg:-





6B) Inorder Traversal :-

```
Void inorder (struct node *root)
{
    if (root != NULL)
    {
        inorder (root -> left)
        Printf("%d", root -> data)
        inorder (root -> right)
    }
}

void preorder(struct node *root)
{
    if (root != NULL)
    {
        printf("%d", root -> data)
        preorder( root -> left)
        preorder(root -> right)
    }
}

void postorder (struct node *root)
{
    if (root != NULL)
    {
        postorder ( root -> left);
        Postorder (root -> right);
        printf("%d", root -> data);
    }
}
```

7a) Insert an element into BST

```
Struct node * insert (struct node *root, int item)
{   if (root == NULL)
    {   root = (struct node *) malloc (sizeof (struct node));
        root -> left = root -> right = NULL;
        root -> data = item;
    }
```

Scanned with CamScanner

```
else if (item < root -> data)
root -> left = insert (root -> left, item);
else if (item > root -> data)
root -> right = insert (root -> right, item);
else
printf ("Duplicate found not allowed");
return root;
}
```

ii) <u>Recursive search</u> :-

```
struct node * search (struct node *root, key)
{     if (root == NULL)
      {  printf (" key not found");
      }
   else if (key < root -> data)
      return search (root -> left, key);
   else if (key > root -> data)
      return search (root -> right, key);
   else
   pf (" key found");
   return root;
}
```

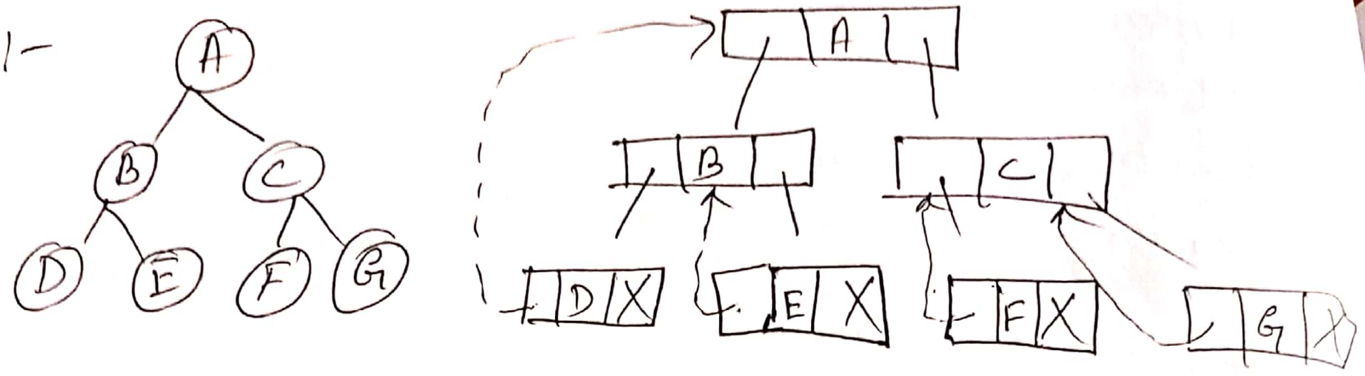
6a) <u>Threaded Binary Tree</u> :- In which NULL pointers are replaced by references to other nodes in the tree called Threads.

Two types i) Left-in. ii) Right-in.

All left child pointers that are null points to ~~its~~ inorder predecessor.

All Right child pointers that are null points to its inorder ~~pre~~ successor.

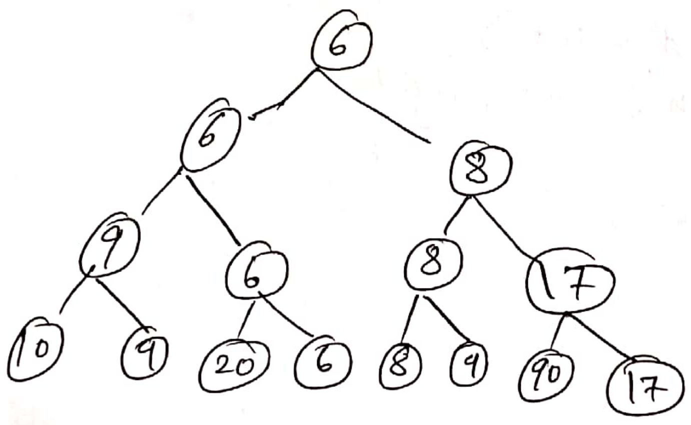
eg:-



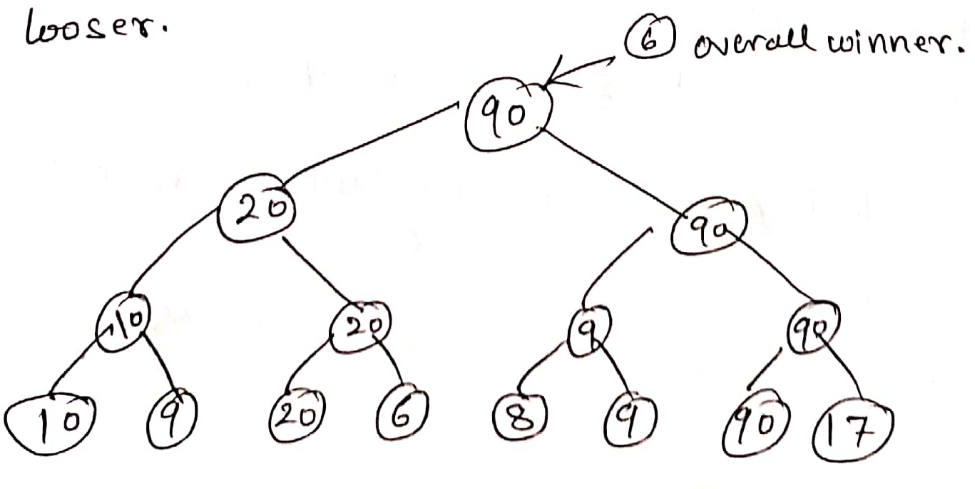
7b) <u>Selection tree</u> :- It is also called as Tournament Tree. This is such a tree data structure using which the winner of a knock out tournament can be selected.

There are two types of trees.

i) <u>Winner Tree</u> :- It is a complete binary Tree in which each node represents smaller of its two children. Thus root node represents the smallest node in the tree.



ii) <u>Looser Tree</u> :- In which non-leaf node retains a pointer to the looser.

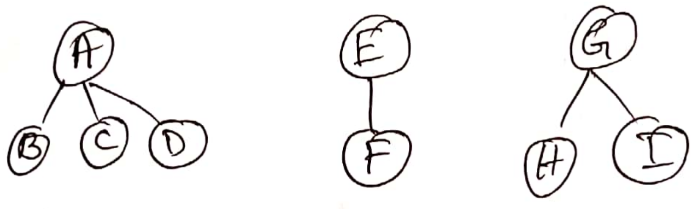


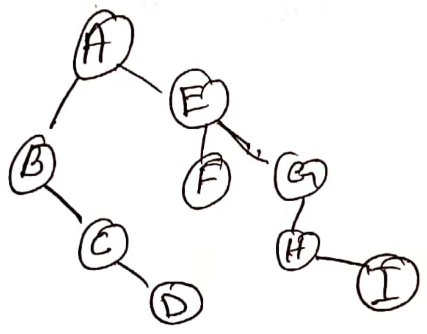
7C.) <u>Transforming forest into Binary Tree</u> :-

if $T_1, T_2, \cdots T_n$ is a forest trees, then binary tree corresponding to this forest, denoted by $B(T_1 T_2 \ldots T_n)$

1) is empty if $n=0$.
2) has root equal to root (T_1)
3) has ~~root~~ left subtree equal to $B(T_{11}, T_{12} \cdots T_{1m})$
4) has right subtree $B(T_2 \cdots T_n)$.

eg:-





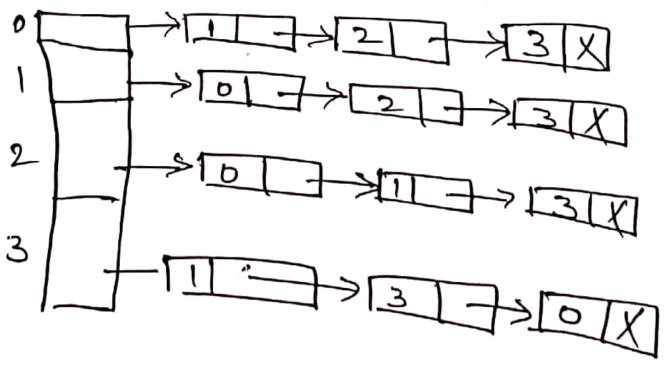
8a) Graph is a pair of sets (V, E) where V is the vertex & E is the set of Edges connecting the pairs of vertices.

$G = (V, E)$.

<u>Adj Matrix Representation.</u>

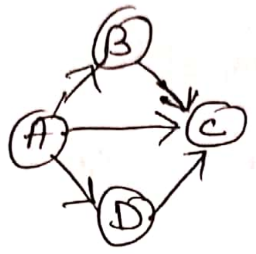
$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 2 & 1 & 1 & 0 & 1 \\ 3 & 1 & 1 & 1 & 0 \end{array}$$

<u>Adj. Linked List</u>



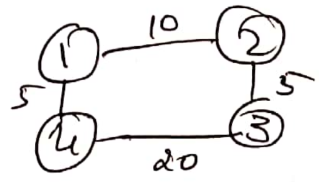
8b) **Digraph** :- A graph whose edges are directed.

eg :-



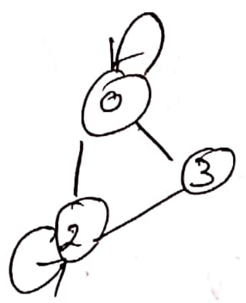
ii) **Weighted Graph** :- Edges have a weight. It typically shows cost of traversing.

eg :-

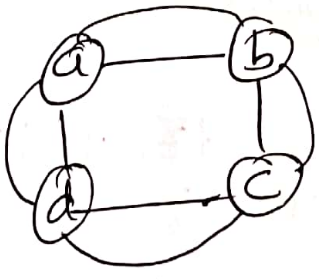


iii) **Self loop** :- It is an edge which starts and ends are the same vertex.

eg :-



iv) **Parallel Edges** :- It refers to the multiple edges that Connect the same pair of vertices in a graph.



8c) Graph operations :-

BFS :- It traverse a graph in breadthward direction & uses a queue to remember to get the next vertex to start a search.

Alg :- S1 - Visit the adj unvisited vertex, Mark it as visited, Display it. Insert it in a Queue.

S2 :- If no adj vertex is found, remove the first vertex from the queue.

S3 :- Repeate S1 & S2 untill Queue is empty.

DFS :- It traverse a graph in depthward direction & uses a stack to remember to get the next vertex to start a search

Alg :- S1 :- Visit the adj unvisited vertex. Mark it as visited Display it, Push it in a stack.

S2 :- If no adj. vertexfound, pop out from stack.

S3 :- Repeate S1 & S2 until the stack is empty.

9a) Collision resolution is the process of handling situations where two or more keys hash to the same index in a hash table.

Methods are

→ Separate Chaining.

→ Open Addressing.

* Linear Probing :- If the calculated index is occupied, move to the next index.
Keep probing sequentially until an empty slot is found

function : $h'(k, i) = [h(k) + i] \bmod m$

here i is the probenumber, m is the table size

eg :- Given keys 12, 13, 10, 20, table size 10

0	10
1	20
2	12
3	13
4	
5	
6	
7	
8	18
9	

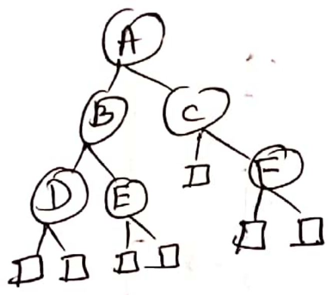
9b) Static hashing :- Here table size is fixed throughout the lifetime of the hash table.
It contains an array of fixed size, where each element in the array is called bucket or slot.

The hash function maps keys to indices in this array.
If two keys hash to the same index, a collision occurs.

Dynamic hashing :- Here size of the hash table can change dynamically based on the number of elements stored
→ It involves extendible hashing. and directory less hashing.
- In extendible, initially it contains small slots.
- Directories :- stores the address of the buckets
- Buckets :- Used to hash the actual data
- Global depth :- Denote number of bits which are used by hash function

i) Leftist Tree :- A leftist tree is a binary tree such that if it is not empty, then shortest (leftchild (x) >= shortest (rightchild(x)) for every internal node x.

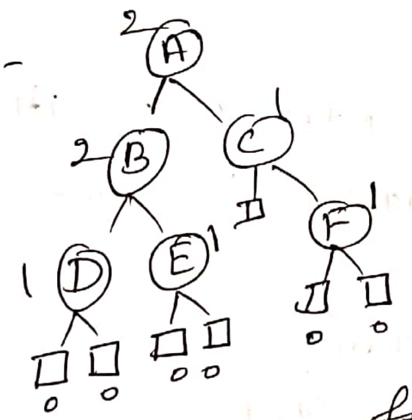
eg:-



Height-Biased :- Let leftchild(x) & right child(x). Then shortest denotes left & right children of internal node x. Then shortest path from x to external node is as follows

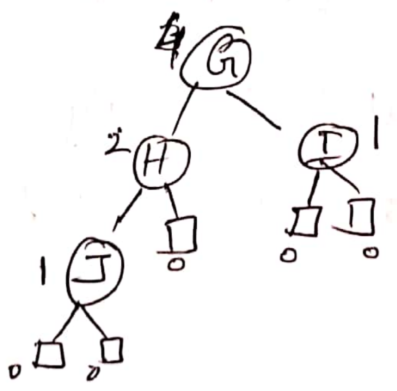
$$\text{shortest}(x) = \begin{cases} 0 & \text{if } x \text{ is an external node} \\ 1 + \min\{\text{shortest}(\text{leftchild}(x)), \text{shortest}(\text{right}(x))\} \end{cases}$$

eg:-



* Weight - Biased :- iff at every internal node w value of the left child is greater than or equal to the w value of right child.

eg :-



10a) <u>Hash functions :-</u>

10b)

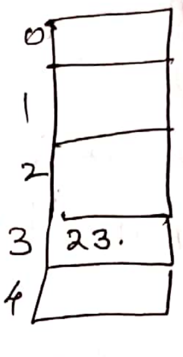
i) <u>Division Method :-</u> we use modular arithmetic system to divide the key value by some integer divisor m.

eg :- Given key = 23, size = ~~0~~5 then

H(x) = key % size

= 23 % ~~0~~5

= 3



ii) <u>Midsquare Method :-</u> We square the value of a key and take the number of digits required to form an address, from the middle position of squared value.

eg :- key is 16 then square is 256.
if we want address two digits then we select the address as <u>56</u>

iii) <u>Folding Method :-</u> The key is actually partitioned into number of parts, each part having the same length as that of the required address.

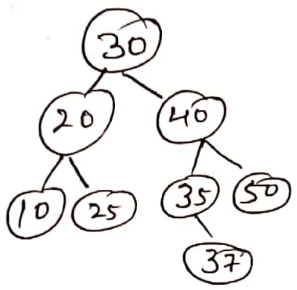
There are two ways i) Fold-shifting
ii) Fold boundary

eg :- 12345678 ⇒ 12 + 34 + 56 + 78 = 180

iv) <u>Digit Analysis :-</u> statistical analysis of digits of the key and select those digits which occur quite frequently.

An AVL tree is a self-balancing binary tree
It maintains a height balance property such that the height difference between the left & right subtrees of any node is at most 1.

eg:-



→ The balance factors of all nodes are within the range of -1 to 1

→ The tree is balanced, & the height of the tree remains logarithmic.

→ The tree maintains the binary search tree property. (The all nodes in the left subtree are less than the root & all nodes in the right subtree are greater than root.)

```
struct node *insert(struct node * nd, int key)
{
    if (nd == NULL)
        return crt(key);
    if (key < node->key)
        node->left = insert(node->left, key);
    elseif (key > node->key)
        node->right = insert(node->right, key);
    node->height = 1+ max(getheight(node->left), getheight(node->
                                                       right));
    int bf = getBalanceFactor(node);
}.
```

13

10c) Red-black Tree :- It is a self-balancing binary search Tree with the following properties

i) Binary Search Tree :- All nodes in the left sub tree have value less than node's value

ii) All nodes in the right subtree have values greater than the nodes value

Color : Each node in this tree is either red or black

root :- Always black.

Splay Tree :- It is a self-adjusting binary search tree with the property that recently accessed elements are quick to access again.

```
    void insert(node ** root, int data)
    {
        node * z = crt(data);
        node * y = NULL;
        node * x = *root;

       while (x != NULL)
        {
          y=x;
          if (z -> data < x -> data)
           {
             x = x -> left;
           }
           else
           {
             x = x -> right;
           }
         }
```

```
z-> parent = y;
if (y == NULL)
{
        *root = z;
  }
  else if (z->data < y->data)
   {
          y->left = z;
     }
     else
      {
          y->right = z;
       }
     fixInsertion(root, z);
};
```

[illegible]

[illegible]

HOD
Computer Science & Engineering
KLS Vishwanathrao Deshpande
Institute of Technology, Haliyal.