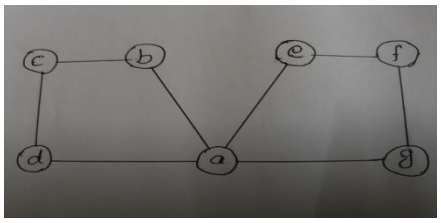
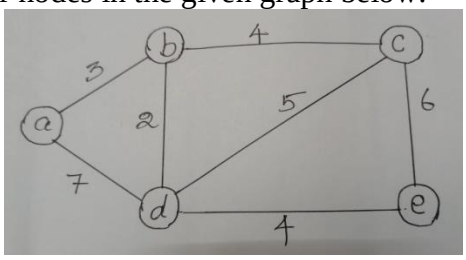
Scheme of Evaluation

Internal Assessment Test 2 -August 2025

Sub:	Data Structures and Algorithms							Sub Code:	MMC203
Date:	05-08-25	Duration:	90 mins	Max Marks:	50	Sem:	II	Branch:	MCA

Q.NO	Description	Marks Distribution	Max Marks
1	Write an algorithm for merge sort with example and also develop a c program for merge sort. • Explanation of Merge sort algorithm with example • Write the C Program code for merge sort	[5] [5]	10
2	Write an algorithm for bubble sort with example and also write a c program for bubble sort. • Explanation of bubble sort algorithm with example • Write the C Program code for bubble sort	[5] [5]	10
3	Write a c program to implement the following searching techniques using recursion a) Linear search b) Binary search • Write the C Program code for Linear & Binary search	[10]	10
4	Write an algorithm for quick sort & analyze with example and also explain with c code • Explanation of quick algorithm with example • Write the C Program code for quick sort	[5] [5]	10
5	Construct the binary search tree for the data 27,5,36,47,19,250,21,44,6. Perform preorder, Inorder and postorder traversal for the constructed BST • Construct the binary search tree for the given data • Findout the traversal for the constructed BST	[5] [5]	10
6	What is Collision? Explain the Collision resolution techniques with proper examples. • Definition of collision • Explanation of the Collision techniques with proper examples	[2] [8]	10

7	Write DFS graph traversal algorithm and write a trace for the following given graph below:  • Explanation of DFS traversal algorithm. • Find out the solution of the given graph	**[5]** **[5]**	**10**
8	Obtain the shortest distance and shortest path from "a" node to all other nodes in the given graph below:  • Find out the solution of the given graph using Dijikstra's algorithm	**[10]**	**10**
9	What is binary tree? Write a note on threaded binary tree with neat diagram. • Definition of Binary tree with example • Explanation of threaded binary tree with neat diagram	**[5]** **[5]**	**10**
10	Write recursive function for the following operations on binary search tree:a) Insert key in BST b) Search key in BST • Write the C Program code for Insert key & Search key in Binary search Tree	**[10]**	**10**

Internal Assessment Test 2 – August 2025

Data Structures and Algorithms							Sub Code:	MMC203
05/08/2025	Duration:	90 min's	Max Marks:	50	Sem:	II	Branch:	MCA

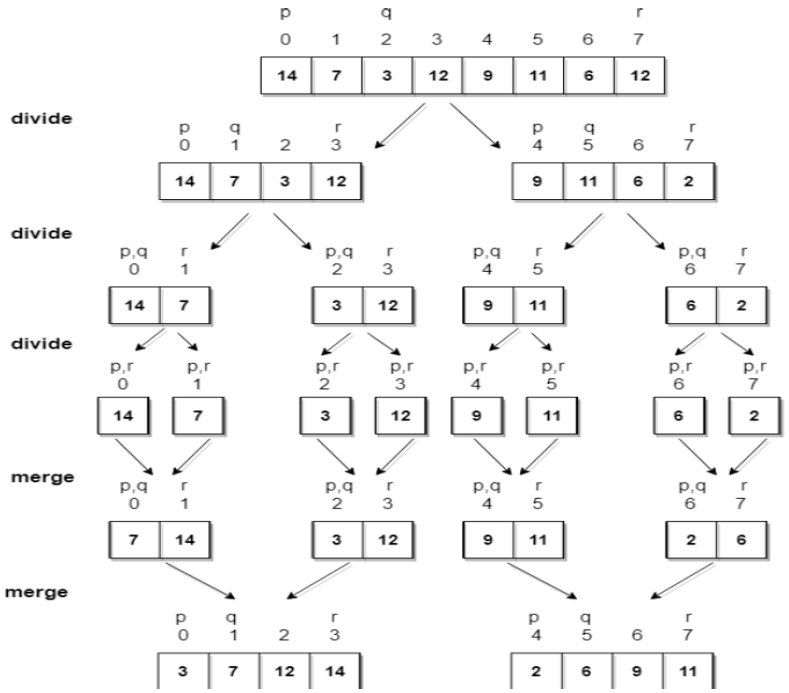
PART I

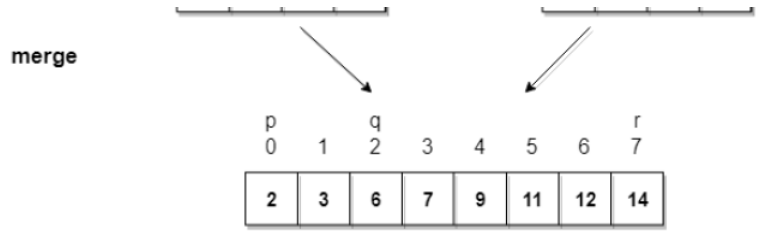
1. **Write an algorithm for merge sort with example and also develop a c program for merge sort.**

   In merge sort, we break the given array midway, for example if the original array had 6 elements, then merge sort will break it down into two sub arrays with 3 elements each. But breaking the original array into 2 smaller sub arrays is not helping us in sorting the array. So we will break these sub arrays into even smaller sub arrays, until we have multiple sub arrays with **single element** in them. Now, the idea here is that an array with a single element is already sorted, so once we break the original array into sub arrays which has only a single element, we have successfully broken down our problem into base problems. And then we have to merge all these sorted sub arrays, step by step to form one single sorted array.

   Let's consider an array with values {14, 7, 3, 12, 9, 11, 6, 12}

   Below, we have a pictorial representation of how merge sort will sort the given array.





In merge sort we follow the following steps:
1. We take a variable p and store the starting index of our array in this. And we take another variable r and store the last index of array in it.
2. Then we find the middle of the array using the formula (p + r)/2 and mark the middle index as q, and break the array into two sub arrays, from p to q and from q + 1 to r index.
3. Then we divide these 2 sub arrays again, just like we divided our main array and this continues.
4. Once we have divided the main array into sub arrays with single elements, then we start merging the sub arrays.

C program for merge sort

```
#include <stdio.h>

// Function to merge two subarrays
void merge(int arr[], int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;  // size of left subarray
    int n2 = right - mid;     // size of right subarray

    int L[n1], R[n2]; // temp arrays

    // Copy data to temp arrays
    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    // Merge the temp arrays back into arr[]
    i = 0; j = 0; k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
```

```
        } else {
            arr[k++] = R[j++];
        }
    }

    // Copy remaining elements of L[], if any
    while (i < n1) {
        arr[k++] = L[i++];
    }

    // Copy remaining elements of R[], if any
    while (j < n2) {
        arr[k++] = R[j++];
    }
}

// Merge sort function
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;

        // Sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    printf("Sorted array: ");
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Main function
int main() {
    int arr[100], n;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter %d elements:\n", n);
```

```
    for(int i = 0; i < n; i++)
        scanf("%d", &arr[i]);
     mergeSort(arr, 0, n - 1);
    printArray(arr, n);

    return 0;
}
```

Sample Output;

```
Enter number of elements: 5
Enter 5 elements:
12 11 13 5 6 7
Sorted array: 5 6 11 12 13
```

2. **Write an algorithm for bubble sort with example and also write a c program for bubble sort.**

The bubble sort makes multiple passes through a list. It compares adjacent items and exchanges those that are out of order. Each pass through the list places the next largest value in its proper place. In essence, each item ―bubbles‖ up to the location where it belongs. The shaded items are being compared to see if they are out of order. If there are n items in the list, then there are n - 1n - 1 pairs of items that need to be compared on the first pass. It is important to note that once the largest value in the list is part of a pair, it will continually be moved along until the pass is complete.

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order. It is a stable sorting algorithm with a time complexity of $O(n^2)$ in the average and worst cases – and O(n) in the best case.

Example:

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1**5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4

(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2

(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)

(1**4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)

(1**2 4** 5 8) –> (1 **2 4** 5 8)

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

C program for bubble sort.

```
#include<stdio.h>
#include<stdlib.h>
void display(int a[],int n);
void bubble_sort(int a[],int n);
int main()
{
	int n,choice,i;
	char ch[20];
	printf("Enter no. of elements u want to sort : ");
	scanf("%d",&n);
	int arr[n];
	for(i=0;i<n;i++)
	{
		printf("Enter %d Element : ",i+1);
		scanf("%d",&arr[i]);
	}
	printf("Please select any option Given Below for Sorting : \n");
	while(1)
	{
		printf("\n1. Bubble Sort\n 2. Display Array.\n3. Exit the Program.\n");
		printf("\nEnter your Choice : ");
		scanf("%d",&choice);
		switch(choice)
		{
			case 1:bubble_sort(arr,n);
				break;
			case 2:display(arr,n);
				break;
			case 3:
				return 0;
			default:
				printf("\nPlease Select only 1-3 option ----\n");
		}
	}
	return 0;
}
//-----------End of main function----------------------

//-------------------Display Function-----------------
void display(int arr[],int n)
{
```

```
        for(int i=0;i<n;i++)
        {
                printf(" %d ",arr[i]);
        }
}
//----------------------Bubble Sort Function-----------
void bubble_sort(int arr[],int n)
{
        int i,j,temp;
        for(i=0;i<n;i++)
        {
                for(j=0;j<n-i-1;j++)
                {
                        if(arr[j]>arr[j+1])
                        {
                                temp=arr[j];
                                arr[j]=arr[j+1];
                                arr[j+1]=temp;
                        }
                }
        }
        printf("After Bubble sort Elements are : ");
        display(arr,n);
}
```

Sample Output:

Enter no. of elements u want to sort : 5
Enter 1 Element : 34
Enter 2 Element : 78
Enter 3 Element : 54
Enter 4 Element : 98
Enter 5 Element : 90
Please select any option Given Below for Sorting :
1. Bubble Sort 2. Display Array.3. Exit the Program.
Enter your Choice : 1
After Bubble sort Elements are : 34 54 78 90 98
1. Bubble Sort
2. Display Array.
3. Exit the Program.

PART II

3. **Write a c program to implement the following searching techniques using recursion a) Linear search b) Binary search**

```
#include <stdio.h>
#include <stdlib.h>
main()
{
/* Declare variables - array_of_number,search_key,i,j,low,high*/
        int array[100],search_key,i,j,n,low,high,location,choice;
        void linear_search(int search_key,int array[100],int n);
        void binary_search(int search_key,int array[100],int n);
        clrscr();
/* read the elements of array */
        printf("ENTER THE SIZE OF THE ARRAY:");
        scanf("%d",&n);
        printf("ENTER THE ELEMENTS OF THE ARRAY:\n");
        for(i=1;i<=n;i++)
        {
                scanf("%d",&array[i]);
        }
/* Get the Search Key element for Linear Search */
        printf("ENTER THE SEARCH KEY:");
        scanf("%d",&search_key);
/* Choice of Search Algorithm */
        printf("_____\n");
        printf("1.LINEAR SEARCH\n");

        printf("2.BINARY SEARCH\n");
        printf("_____\n");
        printf("ENTER YOUR CHOICE:");
        scanf("%d",&choice);


        switch(choice)
        {
                case 1:
                        linear_search(search_key,array,n);
                        break;
                case 2:
                        binary_search(search_key,array,n);
                        break;
                default:
                        exit(0);
        }
        getch();
        return 0;
```

```
}
/* LINEAR SEARCH */
void linear_search(int search_key,int array[100],int n)
{
/*Declare Variable */
        int i,location;
        for(i=1;i<=n;i++)
        {
                if(search_key == array[i])
                {
                        location = i;
                        printf("_____\n");
                        printf("The location of Search Key = %d is
%d\n",search_key,location);
                        printf("_____\n");
                }
        }
}
/* Binary Search to find Search Key */
void binary_search(int search_key,int array[100],int n)
{
        int mid,i,low,high;
        low = 1;
        high = n;
        mid = (low + high)/2;
        i=1;
        while(search_key != array[mid])
        {
                if(search_key<= array[mid])
                {
                        low = 1;
                        high = mid+1;
                        mid = (low+high)/2;
                }
                else
                {
                        low = mid+1;
                        high = n;
                        mid = (low+high)/2;
                }
        }
        printf("_____\n");
        printf("location=%d\t",mid);
        printf("Search_Key=%d Found!\n",search_key);
        printf("_____\n");
}
```

Output:

```
ENTER THE SIZE OF THE ARRAY:6
ENTER THE ELEMENTS OF THE ARRAY:
45      6       86      23      64      77
ENTER THE SEARCH KEY:86
_____________________
1. LINEAR SEARCH
2. BINARY SEARCH
_____________________
ENTER YOUR CHOICE:2
_____________________________________________
Location =3
Search Key = 86 Found!
```

4. Write an algorithm for quick sort & analyze with example and also explain with c code

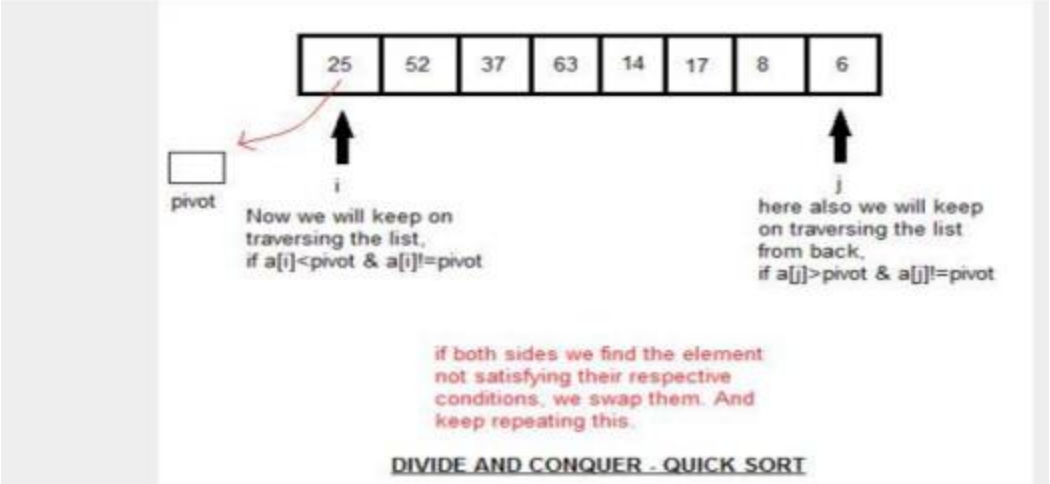
Quick Sort, as the name suggests, sorts any list very quickly. Quick sort is not stable search, but it is
very fast and requires very less aditional space. It is based on the rule of Divide and Conquer(also called
partition-exchange sort). This algorithm divides the list into three main parts :

1. Elements less than the Pivot element
2. Pivot element
3. Elements greater than the pivot element

In the list of elements, mentioned in below example, we have taken 25 as pivot. So after the first pass, the
list will be changed like this.

6 8 17 14 25 63 37 52

Hence after the first pass, pivot will be set at its position, with all the elements smaller to it on its left and
all the elements larger than it on the right. Now 6 8 17 14 and 63 37 52 are considered as two separate lists,
and same logic is applied on them, and we keep doing this until the complete list is sorted.



How Quick Sorting Works

```
void qsort(int arr[], int left, int right)
{
int i,j,pivot,tmp;
if(left<right)
{
pivot=left;
i=left+1;
j=right;
while(i<j)
{
while(arr[i]<=arr[pivot] && i<right)
i++;
while(arr[j]>arr[pivot])
j--;
if(i<j)
{
tmp=arr[i];
arr[i]=arr[j];
arr[j]=tmp;
}
}
tmp=arr[pivot];
arr[pivot]=arr[j];
arr[j]=tmp;
qsort(arr,left,j-1);
```

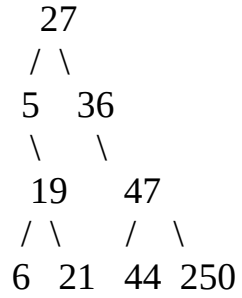
```
qsort(arr,j+1,right);
}
}
```

Complexity Analysis of Quick Sort
Worst Case Time Complexity : O(n2
)
Best Case Time Complexity : O(n log n)
Average Time Complexity : O(n log n)
Space Complexity : O(n log n)
Space required by quick sort is very less, only O(n log n) additional space is required.
Quick sort is not a stable sorting technique, so it might change the occurence of two similar elements
in the list while sorting.

PART III

5. **Construct the binary search tree for the data 27,5,36,47,19,250,21,44,6. Perform preorder, Inorder and postorder traversal for the constructed BST**

BST Structure



Step 2: Traversals

Preorder (Root → Left → Right)

```
27, 5, 19, 6, 21, 36, 47, 44, 250
```

Inorder (Left → Root → Right)

(For BST, this gives sorted order)

```
5, 6, 19, 21, 27, 36, 44, 47, 250
```

Post order (Left → Right → Root)

6, 21, 19, 5, 44, 250, 47, 36, 27

6. What is Collision? Explain the Collision resolution techniques with proper examples.

The situation in which the hash function returns the same hash key (home bucket) for more than one record is called collision and two same hash keys returned for different records is called synonym.

COLLISION RESOLUTION TECHNIQUES

If collision occurs then it should be handled by applying some techniques. Such a technique is called collision handling technique.

1.Chaining
2.Open addressing (linear probing)
3.Quadratic probing
4.Double hashing
6.Rehashing

CHAINING

In collision handling method chaining is a concept which introduces an additional field with data i.e. chain. A separate chain table is maintained for colliding data. When collision occurs then a linked list(chain) is maintained at the home bucket.

For eg;

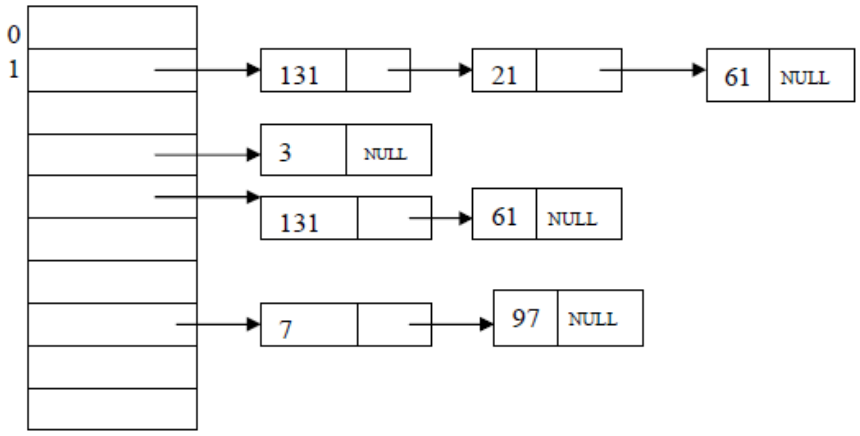
Consider the keys to be placed in their home buckets are

131, 3, 4, 21, 61, 7, 97, 8, 9

then we will apply a hash function as H(key) = key % D

Where D is the size of table. The hash table will be

Here D = 10



A chain is maintained for colliding elements. for instance 131 has a home bucket (key) 1. similarly key 21 and 61 demand for home bucket 1. Hence a chain is maintained at index 1.

OPEN ADDRESSING – LINEAR PROBING

This is the easiest method of handling collision. When collision occurs i.e. when two records demand for the same home bucket in the hash table then collision can be solved by placing the second record linearly down whenever the empty bucket is found. When use linear probing (open addressing), the hash table is represented as a one-dimensional array with indices that range from 0 to the desired table size-1. Before inserting any elements into this table, we must initialize the table to represent the situation where all slots are empty. This allows us to detect overflows and collisions when we inset elements into the table. Then using some suitable hash function the element can be inserted into the hash table.

For example:

Consider that following keys are to be inserted in the hash table

131, 4, 8, 7, 21, 5, 31, 61, 9, 29

Initially, we will put the following keys in the hash table.
We will use Division hash function. That means the keys are placed using the formula

H(key) = key % tablesize
H(key) = key % 10

For instance the element 131 can be placed at

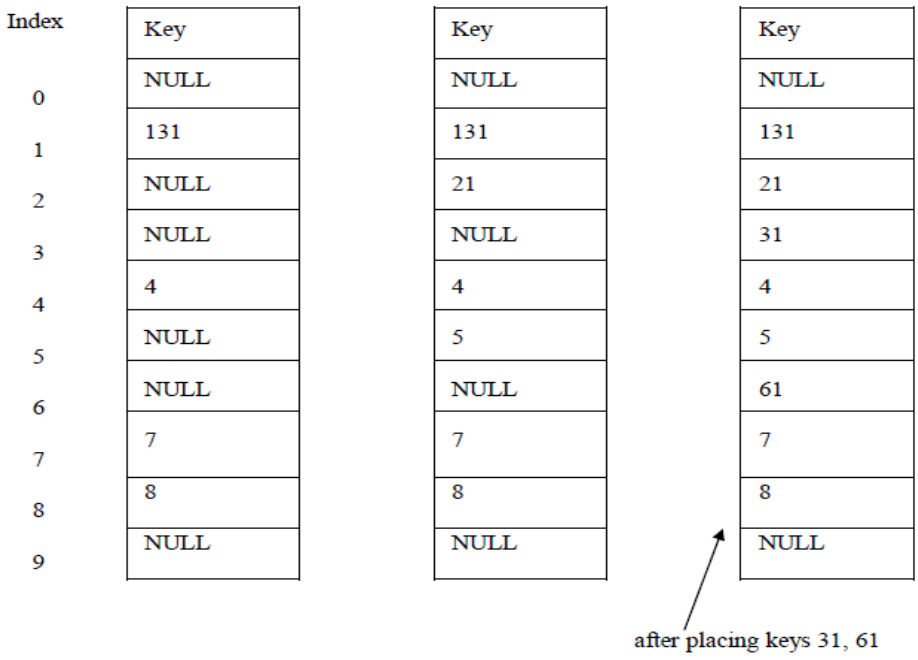
H(key) = 131 % 10
= 1

Index 1 will be the home bucket for 131. Continuing in this fashion we will place 4, 8, 7.

Now the next key to be inserted is 21. According to the hash function

H(key)=21%10
H(key) = 1

But the index 1 location is already occupied by 131 i.e. collision occurs. To resolve this collision we will linearly move down and at the next empty location we will prob the element. Therefore 21 will be placed at the index 2. If the next element is 5 then we get the home bucket for 5 as index 5 and this bucket is empty so we will put the element 5 at index 5.



The next record key is 9. According to decision hash function it demands for the home bucket 9. Hence we will place 9 at index 9. Now the next final record key 29 and it hashes a key 9. But home bucket 9 is already occupied. And there is no next empty bucket as the table size is limited to index 9. The overflow occurs. To handle it we move back to bucket 0 and is the location over there is empty 29 will be placed at 0^{th} index.

Problem with linear probing:

One major problem with linear probing is primary clustering. Primary clustering is a process in which a block of data is formed in the hash table when collision is resolved.

19%10 = 9
18%10 = 8
39%10 = 9
29%10 = 9
8%10 = 8

cluster is formed

rest of the table is empty

Key
39
29
8
18
19

this cluster problem can be solved by quadratic probing.

QUADRATIC PROBING:

Quadratic probing operates by taking the original hash value and adding successive values of an arbitrary quadratic polynomial to the starting value. This method uses following formula.

$$H(key) = (Hash(key) + i^2) \% m)$$

where m can be table size or any prime number.

for eg; If we have to insert following elements in the hash table with table size 10:

37, 90, 55, 22, 17, 49, 87

37 % 10 = 7
90 % 10 = 0
55 % 10 = 5
22 % 10 = 2
11 % 10 = 1

Now if we want to place 17 a collision will occur as 17%10 = 7 and bucket 7 has already an element 37. Hence we will apply quadratic probing to insert this record in the hash table.

0	90
1	11
2	22
3	
4	
5	55
6	
7	37
8	
9	

$H_i(key) = (Hash(key) + i^2) \% m$

Consider i = 0 then
$(17 + 0^2) \% 10 = 7$

$(17 + 1^2)$ % 10 = 8, when i =1

The bucket 8 is empty hence we will place the element at index 8.
Then comes 49 which will be placed at index 9.

49 % 10 = 9

0	90
1	11
2	22
3	
4	
5	55
6	
7	37
8	49
9	

Now to place 87 we will use quadratic probing.

$(87 + 0)$ % 10 = 7
$(87 + 1)$ % 10 = 8… but already occupied
$(87 + 2^2)$ % 10 = 1.. already occupied
$(87 + 3^2)$ % 10 = 6

It is observed that if we want place all the necessary elements in the hash table the size of divisor (m) should be twice as large as total number of elements.

0	90
1	11
2	22
3	
4	
5	55
6	87
7	37
8	49
9	

DOUBLE HASHING

Double hashing is technique in which a second hash function is applied to the key when a collision occurs. By applying the second hash function we will get the number of positions from the point of collision to insert.
There are two important rules to be followed for the second function:

- it must never evaluate to zero.
- must make sure that all cells can be probed.

The formula to be used for double hashing is

H_1(key) = key mod tablesize
H_2(key) = M – (key mod M)

where M is a prime number smaller than the size of the table.

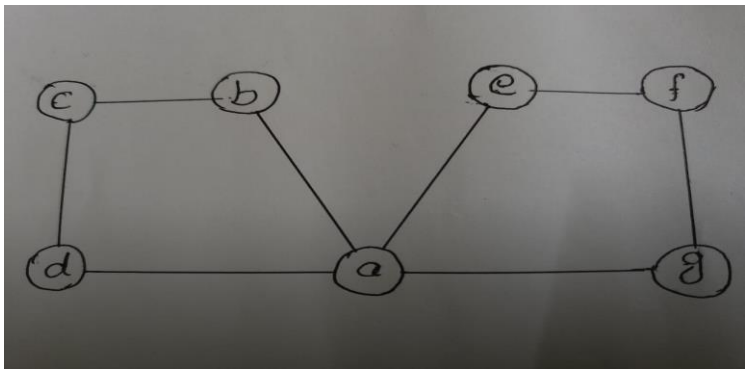
Consider the following elements to be placed in the hash table of size 10
37, 90, 45, 22, 17, 49, 55
Initially insert the elements using the formula for H_1(key).
Insert 37, 90, 45, 22

$H_1(37) = 37$ % 10 = 7
$H_1(90) = 90$ % 10 = 0
$H_1(45) = 45$ % 10 = 5
$H_1(22) = 22$ % 10 = 2
$H_1(49) = 49$ % 10 = 9

Key
90
22
45
37
49

PART IV

7**. Write DFS graph traversal algorithm and write a trace for the following given graph below:**



Solution:

DFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal. We use the following steps to implement DFS traversal...

Step 1 - Define a Stack of size total number of vertices in the graph.

Step 2 - Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

Step 3 - Visit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it on to the stack.

Step 4 - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

Step 5 - When there is no new vertex to visit then use back tracking and pop one vertex from the stack.

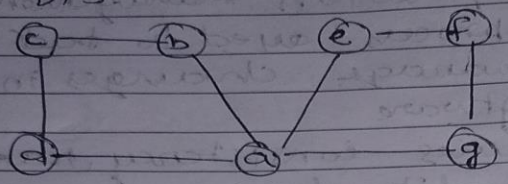
Step 6 - Repeat steps 3, 4 and 5 until stack becomes Empty.

Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

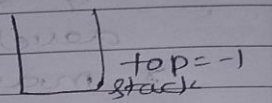
Back tracking is coming back to the vertex from which we reached the current vertex.

PAGE NO :
DATE :

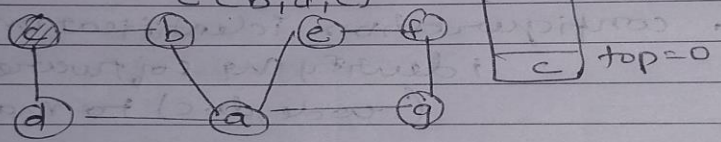
DFS:-



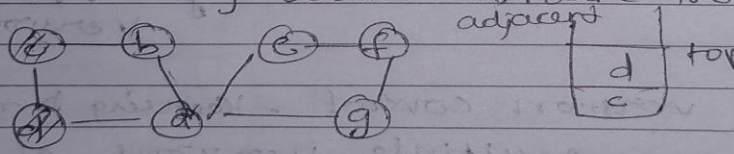
step1:- initialize stack



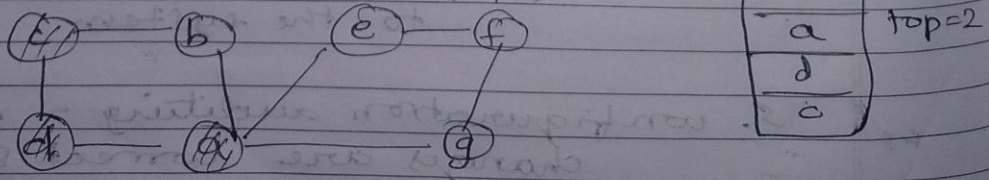
step2:- mark 'c' as visited and push it on to the stack and visit any unvisited node from adjacent c(b,d,e)



step3:- mark 'd' as visited and push it on to the stack and visit any unvisited node from d(a)

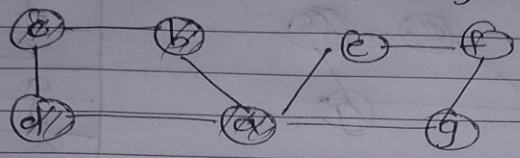


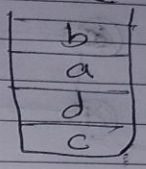
step4:- mark 'a' as visited and push it on to the stack and visit any unvisited adjacent node from a(b,e)



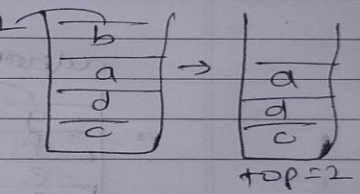
PAGE NO :
DATE :

step 5:- visit 'b' as visited and push it on to stack and visit any unvisited adjacent node of 'b'

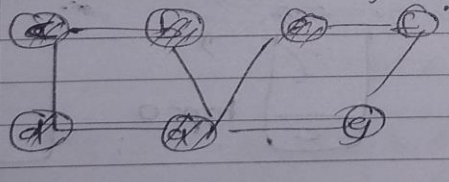


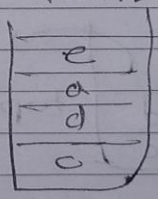


step 6:- since all adjacent vertices of 'b' are visited, pop 'b' and through backtracking visit any adjacent unvisited node from a (e, g).

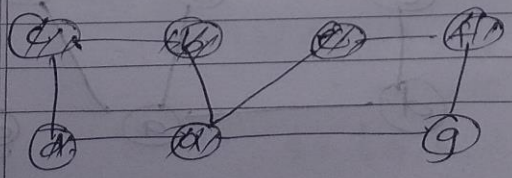


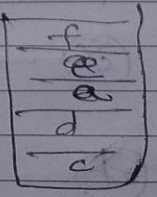
step 7:- mark as visited visit 'e' and push it on to the stack and visit any unvisited adjacent from e(f)





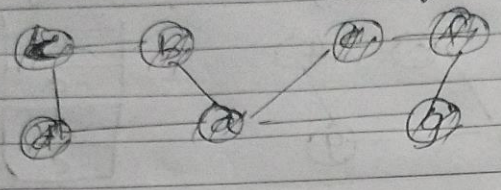
step 8:- mark 'f' as visited and push it on to the stack and visit any unvisited adjacent node from f (g)

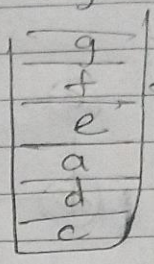




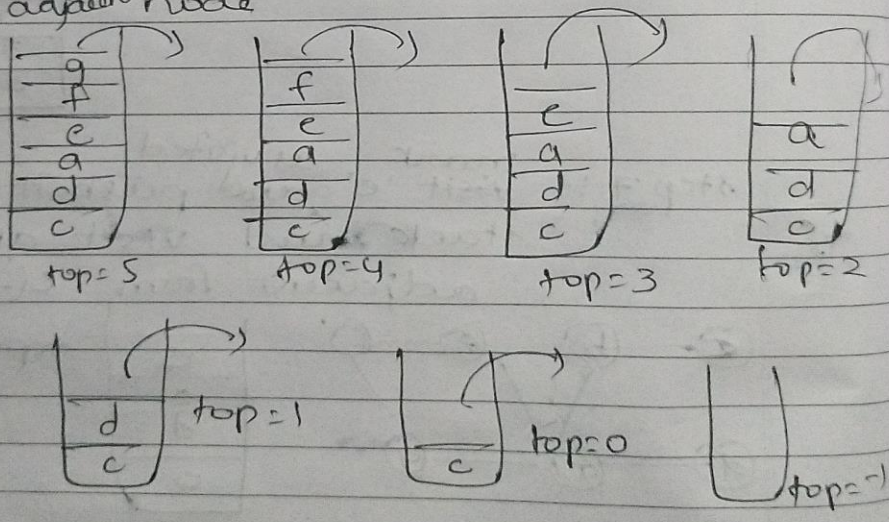
PAGE NO :
DATE :

Step 9:- mark 'g' as visited and visit any unvisited adjacent node of g



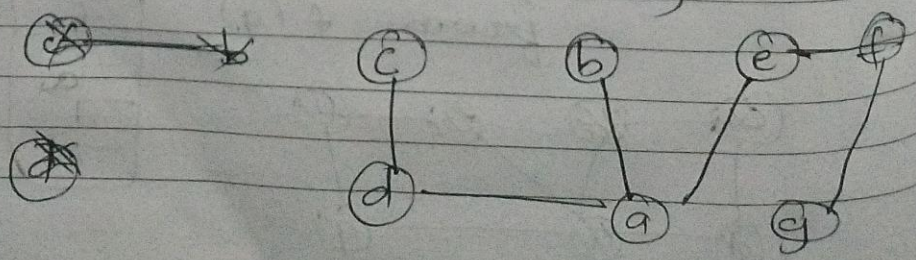


Step 10:- since all nodes are visited and no unvisited nodes left we keep popping ~~the~~ nodes from stack till we find any unvisited adjacent node

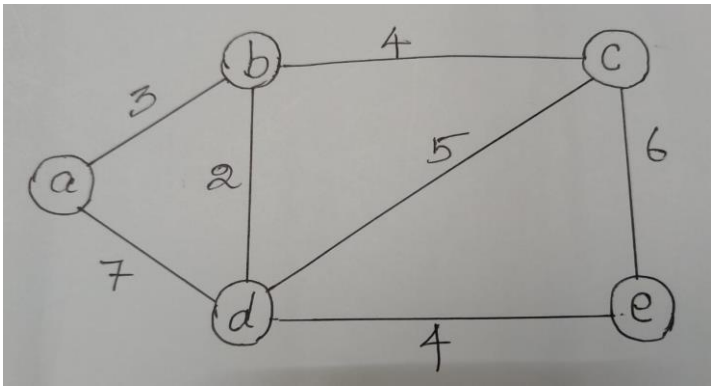


∴ Graph traversal is completed

o/p:- c → d → a → b → e → f → g



8. **Obtain the shortest distance and shortest path from "a" node to all other nodes in the given graph below:**



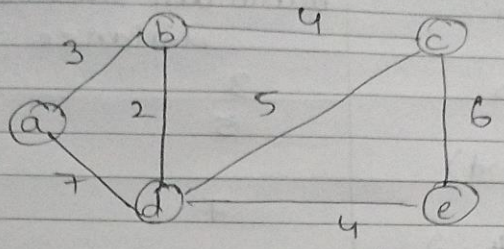
Solution:

Here are the shortest distances from **a** to every node (Dijkstra result) and one shortest path for each:

- **a → a** : distance **0** (path: `a`)
- **a → b** : distance **3** (path: `a → b`)
- **a → d** : distance **5** (path: `a → b → d`)
- **a → c** : distance **7** (path: `a → b → c`)
- **a → e** : distance **9** (path: `a → b → d → e`)

PAGE NO :
DATE :

Dijkstra :-



Step 1 :- Let source node $s = \{a\}$ and

$p = \{b, c, d, e\}$

$d(a,b) = 3$

$d(a,c) = \infty$

$d(a,d) = 7$

$d(a,e) = \infty$

minimum $(3, \infty, 7, \infty) = 3$

∴ choose vertex 'b'

Step 2 = $s = \{a, b\}$ $p = \{c, d, e\}$

$d(a,c) = 7$ $(a \to b \to c)$

$d(a,d) = 5$ $(a \to b \to d)$

$d(a,e) = \infty$

$\min(7, 5, \infty) = 5$

∴ choose vertex 'd'

Step 3 = $s = \{a, b, d\}$ $p = \{c, e\}$

$d(a,c) = 7$ $(a \to b \to c)$

$d(a,e) = 9$ $(a \to d \to e)$

$\min(7, 9) = 7$

∴ choose vertex 'c'

Step 4 : $s = \{a, b, d, c\}$ $p = \{e\}$

$d(a,e) = 9$

∴ choose vertex 'e'

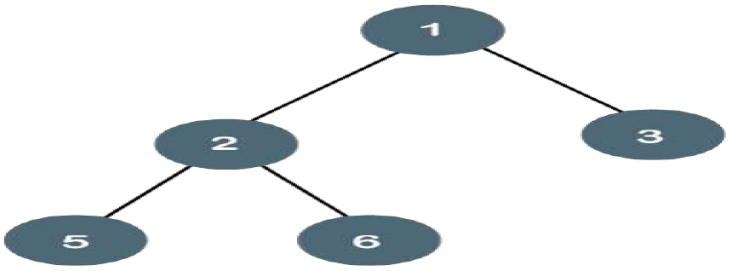
$s = \{a, b, d, c, e\}$

PART V

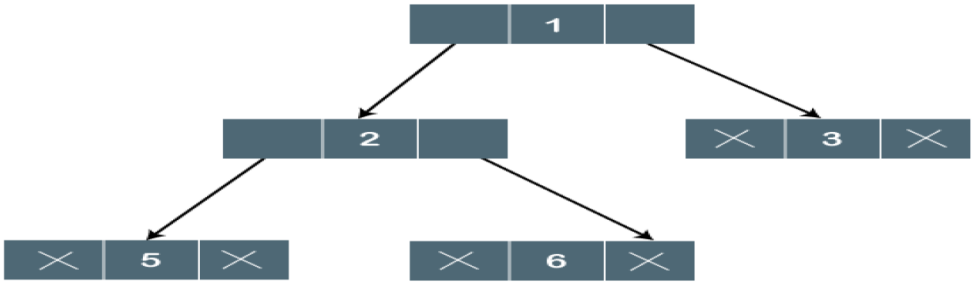
9. What is binary tree? Write a note on threaded binary tree with neat diagram.

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.

Let's understand the binary tree through an example.



The above tree is a binary tree because each node contains the utmost two children. The logical representation of the above tree is given below:



In the above tree, node 1 contains two pointers, i.e., left and a right pointer pointing to the left and right node respectively. The node 2 contains both the nodes (left and right node); therefore, it has two pointers (left and right). The nodes 3, 5 and 6 are the leaf nodes, so all these nodes contain **NULL** pointer on both left and right parts.

Properties of Binary Tree ○ At each level of i, the maximum number of nodes is 2^i. ○ The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to (1+2+4+8) = 15. In general, the maximum number of nodes possible at height h is ($2^0 + 2^1 + 2^2 + \ldots 2^h$) = 2^{h+1} -1. ○ If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum. If there are 'n' number of nodes in the binary tree.

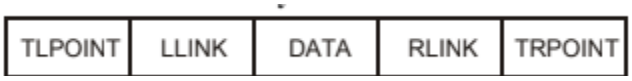
Needed for threaded binary tree

A binary tree with ‗n' nodes need 2n pointers out of which (n + 1) are null pointers. A.J. Perlis and C.Thomton devised a method to utilise these (n + 1) null pointers. These null pointers are now called as threads, which could be effectively used to point to significant nodes choosen according to a traversal scheme to be used for the tree.Threads that take the place of a left child pointer indicate the inorder predecessor, whereas those taking the place of a right child pointer lead to the inorder successor.

Threaded binary tree

Threaded binary tree is the left subtree of a root node whose right child pointer points to itself. Inorder to keep track of which pointers are threads two more additional br fields TLPOINT and TRPOINT are required in each node.

Linked representation of a threaded binary tree

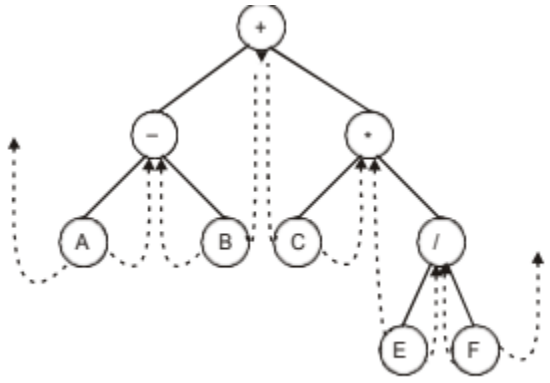


For a node ‗N', if TRPOINT (N) is false then RLINK (N) is a normal pointer.

 If TRPOINT (N) = TRUE then RLINK (N) is a thread pointer, which points to the node which would occur after the node ‗N' during inorder traversal.

Similarly if TLPOINT (N) = False then LLINK (N) is a normal pointer, else LLINK(N) is a thread pointer, which points to the node that would immediately precede the node N, when the binary tree is traversed in inorder.

Example:(A – B) + C * (E/F)



3.7.2: Threaded binary tree for the expression (A – B) + C * (E/F)

Two ways of threading
 One way threading Two way threading
One way threading: Thread appears only on the RLINK of a node, pointing to the inorder successor of the node.
Two way threading: Threads are appear in both the links LLINK and RLINK and points to the inorder predecessor and inorder successor respectively.

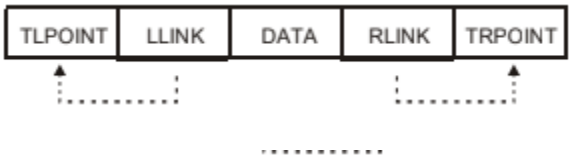
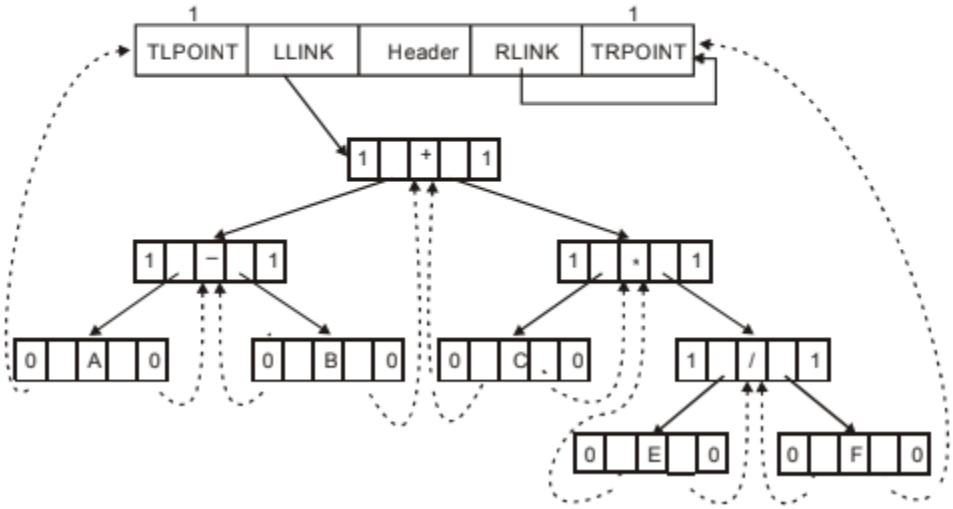


Fig. 3.7.3

An empty threaded binary tree with only.



10. Write recursive function for the following operations on binary search tree:a) Insert key in BST b) Search key in BST

```
#include <stdio.h>
#include <stdlib.h>

// Structure for BST Node
struct Node {
    int key;
    struct Node *left, *right;
};
```

```
// Function to create a new node
struct Node* newNode(int item) {
    struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Recursive function to insert a key in BST
struct Node* insert(struct Node* root, int key) {
    if (root == NULL) // If tree is empty, return new node
        return newNode(key);

    if (key < root->key)       // Insert in left subtree
        root->left = insert(root->left, key);
    else if (key > root->key)  // Insert in right subtree
        root->right = insert(root->right, key);

    return root; // Return unchanged root pointer
}

// Recursive function to search a key in BST
struct Node* search(struct Node* root, int key) {
    if (root == NULL || root->key == key)
        return root;

    if (key < root->key)
        return search(root->left, key);
    else
        return search(root->right, key);
}

// Inorder Traversal (for testing)
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
```

```
}

int main() {
    struct Node* root = NULL;
    int keys[] = {27, 5, 36, 47, 19, 250, 21, 44, 6};
    int n = sizeof(keys) / sizeof(keys[0]);

    // Insert keys into BST
    for (int i = 0; i < n; i++) {
        root = insert(root, keys[i]);
    }

    printf("Inorder Traversal of BST: ");
    inorder(root);
    printf("\n");

    // Search for a key
    int searchKey = 44;
    struct Node* result = search(root, searchKey);
    if (result != NULL)
        printf("Key %d found in BST.\n", searchKey);
    else
        printf("Key %d not found in BST.\n", searchKey);

    return 0;
}
```

```
Inorder Traversal of BST: 5 6 19 21 27 36 44 47 250
Key 44 found in BST.
```