CBCS SCHEME

USN | | | | | | | | | | BCS304

Third Semester B.E./B.Tech. Degree Examination, Dec.2024/Jan.2025
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 the classification of data structures with a neat diagram.	8	L2	CO1
	b.	Write a C Functions to implement pop , push and display operations for stacks using assays.	7	L2	CO2
	c.	Differentiate structures and unions.	5	L2	CO1
		OR			
Q.2	a.	Write an algorithm to evaluate a postfix expression and apply the same for the given postfix expression. 6 2 / 3 – 4 2 * +.	7	L3	CO2
	b.	Explain the dynamic memory allocation function in detail.	8	L2	CO1
	c.	What is Sparse matrix? Give the triplet form of a given matrix and find its transpose $A = \begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$	5	L3	CO1
		Module – 2			
Q.3	a.	Define Queue. Discuss how to represent a queue using dynamic assays.	8	L2	CO2
	b.	Write a C Function to implement insertion ( ) , deletion ( ) and display ( ) operations on circular queue.	6	L3	CO2
	c.	Write a note on Multiple stacks and queues with suitable diagram.	6	L2	CO2
		OR			
Q.4	a.	What is a linked list? Explain the different types of linked list with neat diagram.	6	L2	CO3
	b.	Write a C function for the following on singly linked list with example : i) Insert a node of the beginning ii) Delete a node at the front iii) Display.	8	L3	CO3
	c.	Write the C function to add two polynomials.	6	L2	CO3

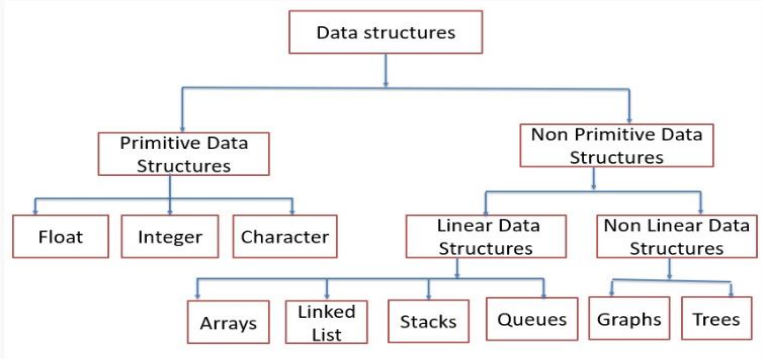
		Module – 3			
Q.5	a.	Discuss how binary trees are represented using : i) Assay ii) Linked list.	6	L2	CO4
	b.	Define Threaded binary tree. Discuss In – threaded binary tree.	6	L2	CO4
	c.	Write the C function for the following additional list operation : i) Inverting Singly linked list ii) Concatenating Singly linked list.	8	L3	CO3
		OR			
Q.6	a.	Discuss Inorder , Preorder , Postorder and Level order traversal with suitable function for each.	8	L3	CO4
	b.	Define the threaded binary tree. Construct threaded binary tree for the following element : A, B, C, D, E, F, G, H, I.	6	L2	CO4
	c.	Write a C function for the following : i) Insert a node at the beginning of doubly linked list. ii) Deleting a node at the end of the doubly linked list.	6	L3	CO3
		Module – 4			
Q.7	a.	Define Forest , Transform the forest into a binary tree and traverse using inorder , preorder and postorder traversal with an example.	8	L1	CO5
	b.	Define Binary search tree. Construct a binary search tree for the following elements : 100 , 85 , 45 , 55 , 120 , 20 , 70 , 90 , 115 , 65 , 130 , 145.	6	L2	CO5
	c.	Discuss Selection tree with an example.	6	L2	CO5
		OR			
Q.8	a.	Define Graph. Explain adjacency matrix and adjacency list representation with an example.	8	L2	CO5
	b.	Define the following terminology with example : i) Digraph ii) Weighted graph iii) Self loop iv) Connected graph.	6	L2	CO5
	c.	Briefly explain about Elementary graph operations.	6	L3	CO5
		Module – 5			
Q.9	a.	Explain in detail about Static and Dynamic Hashing.	6	L2	CO5
	b.	What is Collision? What are the methods to resolve collision?	7	L2	CO5
	c.	Explain Priority queue with the help of an examples.	7	L2	CO5
		OR			
Q.10	a.	Define Hashing. Explain different hashing functions with suitable examples.	12	L2	CO5
	b.	Write short note on : i) Leftist trees ii) Optimal binary search tree.	8	L3	CO5

* * * * *

1a. Define Data structures. With a neat diagram, explain the classification of Data structures with examples. (8 marks)

Diagram – 5 marks
Explanation with Examples – 3 marks

Data structure is a representation of the logical relationships existing between individual elements of data. The logical or mathematical model of a particular organization of data is called a data structure.



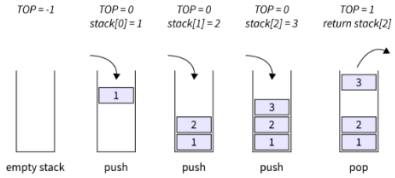
Primitive data structures allow storing only one value at a particular location. Primitive data structures of any programming language are the data types that are predefined in that programming language.

boolean:- The Boolean data type allows storing two values only i.e. true and false. Mostly boolean data type is used for testing the conditions.
char:- The character data type allows you to store a single character. It stores ASCII assigned values of the lower case alphabets, upper case alphabets and some special symbols.
integer:- Integer are used to store the value of the numeric type. It allows for storing both negative and positive values.
float:- The float data type allows you to store the floating value.

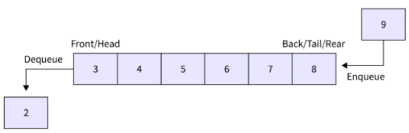
Non-primitive data structures are the data structure created by the programmer with the help of primitive data structures. Non-primitive data structures are divided into **linear and non-linear data structures**

Array:- Array is a linear data structure that stores the elements of similar data types at a contiguous memory location.

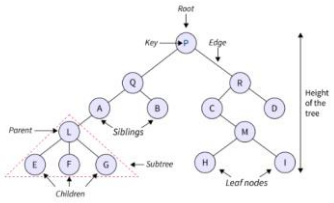
Stack:- Stack is a linear data structure that works on the principle of LIFO(Last In First Out) which allows insertion and deletion of elements from one end only i.e. top.



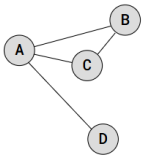
Queue:- Queue is a linear data structure that works on the principle of FIFO(First In First Out) which allows the insertion of elements at one end i.e. rear and deletion of elements from another end i.e. front.



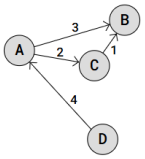
Tree:- Tree is a non-linear data structure that stores the element as nodes in the form of a hierarchical (parent-child) relationship.



Graph: A Graph is a non-linear data structure that consists of vertices (nodes) and edges.



An undirected Graph



A directed and weighted Graph,

1b. Write a C functions to implement pop, push and display operations for stacks using arrays. (7 marks)

Syntax- 2 marks
Logic – 5 marks

```
#define MAX 5
int a[MAX];
int top=-1;
void push(int item)
{
        if(top==(MAX-1))
        {
                printf("STACK OVERFLOW\n");
                return;
        }
        top=top+1;
        a[top]=item;
}
int pop()
{
        if(top==-1)
        {
                printf("STACK UNDERFLOW\n");
                return -1;
        }
        int item=a[top];
        top=top-1;
        return item;
}
void display()
{
        int i;
        if(top==-1)
        {
                printf("STACK IS EMPTY");
                return;



        }
        printf("STACK ELEMENTS ARE:\n");
        for(i=top;i>=0;i--)
        {
                printf("%d ",a[i]);
        }
        printf("'\n");
}
```

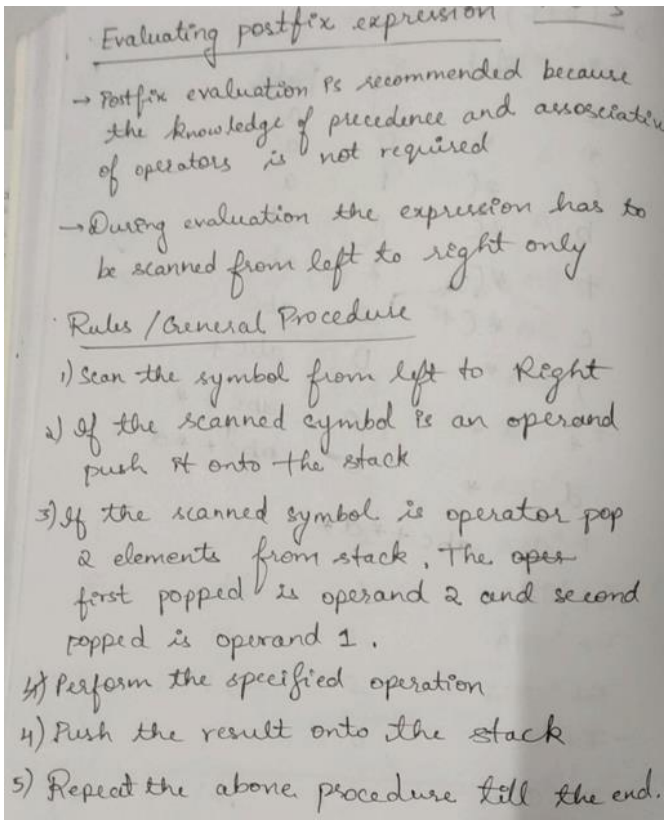
1c. Differentiate structures and unions. (5 marks)

Explanation with keyword – 5 marks

Parameter	Structure	Union
Definition	A structure is a user-defined data type that groups different data types into a single entity.	A union is a user-defined data type that allows storing different data types at the same memory location.
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union
Size	The size is the sum of the sizes of all members, with padding if necessary.	The size is equal to the size of the largest member, with possible padding.
Memory Allocation	Each member within a structure is allocated unique storage area of location.	Memory allocated is shared by individual members of union.
Data Overlap	No data overlap as members are independent.	Full data overlap as members shares the same memory.
Accessing Members	Individual member can be accessed at a time.	Only one member can be accessed at a time.

2a. Write an algorithm to evaluate postfix expression and apply the same for the given postfix expression 6 2 / 3 – 4 2 * +. (7 marks)

Algorithm – 5 marks
evaluating the expression – 2 marks


Evaluating postfix expression

→ Postfix evaluation is recommended because the knowledge of precedence and associativity of operators is not required

→ During evaluation the expression has to be scanned from left to right only

Rules / General Procedure

1) Scan the symbol from left to Right
2) If the scanned symbol is an operand push it onto the stack
3) If the scanned symbol is operator pop 2 elements from stack, The oper- first popped is operand 2 and second popped is operand 1.
4) Perform the specified operation
4) Push the result onto the stack
5) Repeat the above procedure till the end.

Token	Stack [0]	[1]	[2]	Top
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
−	6/2−3			0
4	6/2−3	4		1
2	6/2−3	4	2	2
*	6/2−3	4*2		1
+	6/2−3+4*2			0

2b. Explain the Dynamic memory allocation function in detail (8 marks)

Types – 2 marks
Explanation – 5 marks
Syntax – 1 marks

Dynamic Memory Allocation can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime. C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under <stdlib.h> header file to facilitate dynamic memory allocation in C programming. They are:

- malloc()
- calloc()
- free()
- realloc()

malloc()

The name "malloc" stands for memory allocation. The malloc() function reserves a block of memory of the specified number of bytes. And, it returns a pointer of void which can be casted into pointers of any form.

Syntax of malloc()

```
ptr = (castType*) malloc(size);
```

calloc()

The name "calloc" stands for contiguous allocation. The malloc() function allocates memory and leaves the memory uninitialized, whereas the calloc() function allocates memory and initializes all bits to zero.

Syntax of calloc()

```
ptr = (castType*)calloc(n, size);
```

free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You must explicitly use free() to release the space. This statement frees the space allocated in the memory pointed by ptr.

Syntax of free()

```
free(ptr);
```

realloc()

If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the realloc() function.

Syntax of realloc()

```
ptr = realloc(ptr, x);
```

2c. Define sparse matrix. For the given sparse matrix, give the linked list representation:

$$A = \begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$

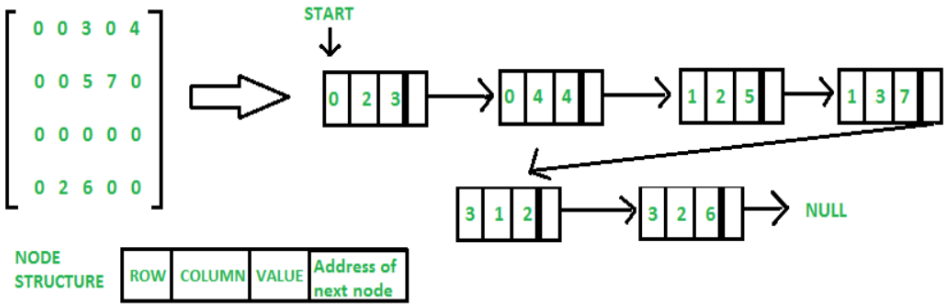
(5marks)

Definition – 2 marks
linked list representation – 3 marks

A **sparse matrix** is a matrix in which most of the elements are zero. This type of matrix is useful in scenarios where storage and computational efficiency are critical, as we can store only the non-zero elements rather than all elements, including the zeros.

In linked list, each node has four fields. These four fields are defined as:

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non-zero element located at index – (row,column)
- **Next node:** Address of the next node.



3a. Define queue. Discuss how to represent a queue using dynamic arrays (8 marks)

Definition – 4 marks
Representation – 4 marks

Queue is a linear data structure that works on the principle of FIFO (First In First Out) which allows the insertion of elements at one end i.e. rear and deletion of elements from another end i.e. front.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10

struct node
{
   int data;
   struct node *link;
```

```
}*front, *rear;

// function protypes
void insert();
void delete();
void queue_size();
void check();
void first_element();

void main()
{
   int choice, value;

   while(1)
   {
      printf("enter the choice \n");
      printf("1 : create an empty queue \n2 : Insert element\n");
      printf("3 : Dequeue an element \n4 : Check if empty\n");
      printf("5. Get the first element of the queue\n");
      printf("6. Get the number of entries in the queue\n");
      printf("7. Exit\n");
      scanf("%d", &choice);
      switch (choice)    // menu driven program
      {
      case 1:
         printf("Empty queue is created with a capacity of %d\n", MAX);
         break;
      case 2:
         insert();
         break;
      case 3:
         delete();
         break;
      case 4:
         check();
         break;
      case 5:
         first_element();
         break;
      case 6:
         queue_size();
         break;
      case 7:
         exit(0);
      default:
         printf("wrong choice\n");
         break;
      }
   }
}

// to insert elements in queue
void insert()
{
   struct node *temp;

   temp = (struct node*)malloc(sizeof(struct node));
   printf("Enter value to be inserted \n");
   scanf("%d", &temp->data);
   temp->link = NULL;
```

```
   if (rear  ==  NULL)
   {
      front = rear = temp;
   }
   else
   {
      rear->link = temp;
      rear = temp;
   }
}

// delete elements from queue
void delete()
{
   struct node *temp;

   temp = front;
   if (front == NULL)
   {
      printf("queue is empty \n");
      front = rear = NULL;
   }
   else
   {
      printf("deleted element is %d\n", front->data);
      front = front->link;
      free(temp);
   }
}

// check if queue is empty or not
void check()
{
   if (front == NULL)
      printf("\nQueue is empty\n");
   else
      printf("*************** Elements are present in the queue *************\n");
}

// returns first element of queue
void first_element()
{
   if (front == NULL)
   {
      printf("**************** The queue is empty ***************\n");
   }
   else
      printf("**************** The front element is %d ***********\n", front->data);
}

// returns number of entries and displays the elements in queue
void queue_size()
{
   struct node *temp;

   temp = front;
   int cnt = 0;
   if (front  ==  NULL)
   {
      printf(" queue empty \n");
```

```
        }
        while (temp)
        {
           printf("%d  ", temp->data);
           temp = temp->link;
           cnt++;
        }
        printf("********* size of queue is %d ******** \n", cnt);
     }
```

3b. Write a C Function to implement Insertion(), deletion() and display() operations on circular queue (6 marks)

Logic – 4 marks
Syntax – 2 marks

```
int f = -1, r = -1;

void insert(char cq[MAX]) {
   char elem;
   printf("\nEnter the element to insert into the queue: ");
   scanf("\n%c", &elem);

   if (f == (r + 1) % MAX) {
      printf("\nQueue Overflow!!!");
      return;
   } else if (f == -1) {
      f = 0;
   }
   r = (r + 1) % MAX;
   cq[r] = elem;
}

void delete(char cq[MAX]) {
   if (f == -1) {
      printf("\nQueue Underflow!!!");
      return;
   }
   printf("\nThe deleted element is: %c", cq[f]);
   if (f == r) {
      f = -1;
      r = -1;
   } else {
      f = (f + 1) % MAX;
   }
}

void display(char cq[MAX]) {
   int i;
   if (f == -1) {
      printf("\nQueue Underflow!!!");

  } else {
     printf("\nThe elements of queue are: ");
     for (i = f; i != r; i = (i + 1) % MAX)
        printf("\t%c", cq[i]);
     printf("\t%c", cq[i]);
  }
}
```

3c.Write a note on Multiple stacks and Queues with suitable diagram. (6 marks)

Explanation – 4 marks
With Example Code – 2 marks

Representing more than two stacks within the same array poses problems since we no longer have an obvious point for the bottom element of each stack. Assuming that we have *n* stacks, we can divide the available memory into *n* segments. This initial division may be done in proportion to the expected sizes of the various stacks, if this is known. Otherwise, we may divide the memory into equal segments.

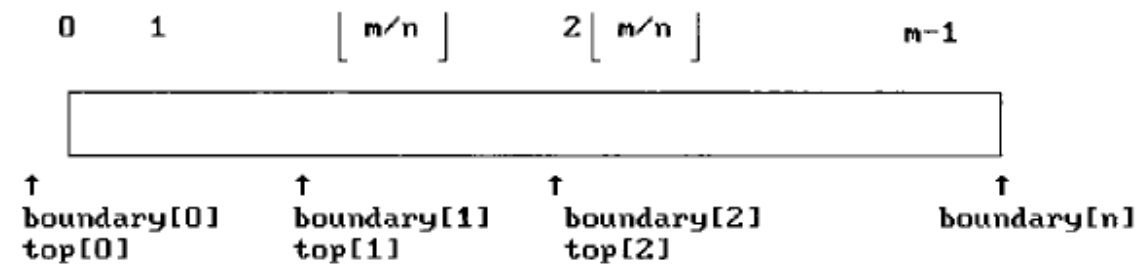
Assume that *stack_no* refers to the stack number of one of the *n* stacks. To establish this stack, we must create indices for both the bottom and top positions of this stack. The bottom element, *boundary*[*stack_no*], $0 \leq stack\_no < MAX\_STACKS$, always points to the position immediately to the left of the bottom element, while *top*[*stack_no*], $0 \leq stack\_no < MAX\_STACKS$ points to the top element. A stack is empty *iff* *boundary*[*stack_no*] = *top*[*stack_no*]. The relevant declarations are:

```
#define MEMORY_SIZE 100 /* size of memory */
#define MAX_STACKS 10 /* max number of stacks plus 1 */
/* global memory declaration */
element memory[MEMORY_SIZE];
int top[MAX_STACKS];
int boundary[MAX_STACKS];
int n;  /* number of stacks entered by the user */
```

To divide the array into roughly equal segments we use the following code:

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

Figure 3.18 shows this initial configuration. In the figure, *n* is the number of stacks entered by the user, $n < MAX\_STACKS$, and $m = MEMORY\_SIZE$. Stack *stack_no* can grow from *boundary*[*stack_no*] + 1 to *boundary*[*stack_no* + 1] before it is full. Since we need a boundary for the last stack, we set *boundary*[*n*] to *MEMORY_SIZE*– 1. Programs 3.12 and 3.13 implement the add and delete operations for this representation.



All stacks are empty and divided into roughly equal segments.

Figure 3.18: Initial configuration for *n* stacks in *memory*[*m*].

4a. What is Linked list? Explain the Different types of linked list with a neat diagram (6 marks)

Explanation-3 marks
Example with Diagrams – 3 marks

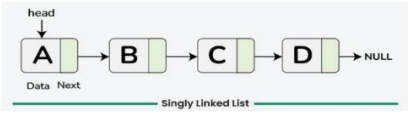
A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers. In simple words, a linked list consists of nodes where each node contains a data field and a reference (link) to the next node in the list.

Types of Linked Lists:

1. Singly Linked List
2. Doubly Linked List
3. Singly Circular Linked List
4. Doubly Circular linked list

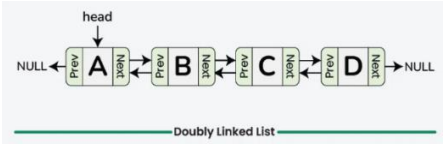
Singly Linked List

Singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type. The node contains a pointer to the next node means that the node stores the address of the next node in the sequence. A single linked list allows the traversal of data only in one way.



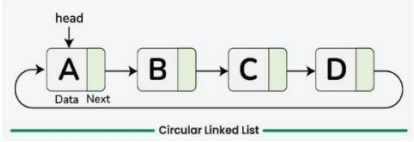
Doubly Linked List

A doubly linked list or a two-way linked list is a more complex type of linked list that contains a pointer to the next as well as the previous node in sequence. Therefore, it contains three parts of data, a pointer to the next node, and a pointer to the previous node. This would enable us to traverse the list in the backward direction as well.



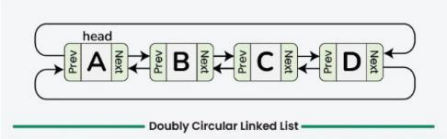
Singly Circular Linked List

A Singly circular linked list is a type of linked list in which the last node's next pointer points back to the first node of the list, creating a circular structure. This design allows for continuous traversal of the list, as there is no null to end the list.



Doubly Circular linked list

Doubly Circular linked list or a circular two-way linked list is a complex type of linked list that contains a pointer to the next as well as the previous node in the sequence



4b. **write the c function for following on singly linked list with example:**

i) Insert a node at beginning
ii) Delete a node at the beginning
iii) Display **(8 marks)**

Logic – 5 marks
Syntax – 3 mark

i) Insert a node at beginning

```
void insert_front() {
   STUDENT node;
        node = create();
        if (start == NULL) {
           start = node;
        } else {
           node->link = start;
           start = node;
        }
     }
```

ii) Delete a node at the beginning

```
     void delete_front() {
        STUDENT temp;
        if (start == NULL) {
           printf("\nList is Empty");
        } else {
           temp = start;
           start = temp->link;
           printf("\nThe deleted student usn is %s", temp->usn);
           free(temp);
        }
     }
```

iii) Display

```
     void status() {
        STUDENT temp;
        int count = 0;
        if (start == NULL) {
           printf("\nList is Empty");
           return;
        }
        temp = start;
        printf("\nThe Student details are: ");
        while (temp != NULL) {
           printf("\n%s\n%s\n%s\n%d\n%ld\n", temp->usn, temp->name, temp->program, temp-
     >sem, temp->phno);
           temp = temp->link;
           count++;
        }
        printf("\nThe number of nodes are: %d", count);
     }
```

4c.Write a C function to add two polynomials (6 marks)

Logic – 3marks
Syntax – 3marks

```
struct polynomial {
   int coeff, x, y, z;
   struct polynomial *link;
};

typedef struct polynomial *POLYNOMIAL;

POLYNOMIAL create() {
   POLYNOMIAL getnode;
   getnode = (POLYNOMIAL)malloc(sizeof(struct polynomial));
   if (getnode == NULL) {
      printf("\nMemory couldn't be allocated!!!");
      return 0;
   }
   return getnode;
}

POLYNOMIAL insert(POLYNOMIAL head, int c, int px, int py, int pz) {
   POLYNOMIAL node, temp;
   node = create();
   node->coeff = c;
   node->x = px;
   node->y = py;
   node->z = pz;
   node->link = NULL;
   temp = head->link;
   while (temp->link != head) { /* Traverse till the end of the list. */
      temp = temp->link;
   }
   temp->link = node;  /* Attach the node to the end of the list. */
   node->link = head;  /* Assign the address of the head to node's link. */
   return head;
}

POLYNOMIAL input_polynomial(POLYNOMIAL head) {
   int i, c, px, py, pz;
   printf("\nEnter 999 to end the polynomial!!!");
   for (i = 1;; i++) {
      printf("\nEnter the coefficient %d: ", i);
      scanf("%d", &c);
      if (c == 999) /* Breaks the loop when 999 is entered indicating end of
input. */
         break;

      printf("\nEnter the power of x: ");
      scanf("%d", &px);
```

```
        printf("\nEnter the power of y: ");
        scanf("%d", &py);
        printf("\nEnter the power of z: ");
        scanf("%d", &pz);
        head = insert(head, c, px, py, pz);
    }
    return head;
}

void display(POLYNOMIAL head) {
    POLYNOMIAL temp;
    if (head->link == head) {
        printf("\nPolynomial doesn't exist!!!");
    } else {
        temp = head->link;
        while (temp != head) {
            printf("%dx^%dy^%dz^%d + ", temp->coeff, temp->x, temp->y, temp-
>z);
            temp = temp->link;
        }
        printf("999");
    }
}

int evaluate_polynomial(POLYNOMIAL head) {
    int vx, vy, vz, sum = 0;
    POLYNOMIAL temp;
    printf("\n\nEnter the value of x, y and z: ");
    scanf("%d%d%d", &vx, &vy, &vz);
    temp = head->link;
    while (temp != head) {
        sum = sum + (temp->coeff * pow(vx, temp->x) * pow(vy, temp->y) *
pow(vz, temp->z));
        temp = temp->link;
    }
    return sum;
}

int main() {
    POLYNOMIAL head;
    int res;
    head = create();
    head->link = head;
    printf("\nEnter the polynomial to be evaluated: ");
    head = input_polynomial(head);
    printf("\nThe given polynomial is: ");
    display(head);
    res = evaluate_polynomial(head);
    printf("\nThe result after evaluation is: %d", res);
   }
```

5a. Discuss how binary trees are represented using: i) Array ii) Linked List (6 marks)

Array Representation – 3 marks
Linked list representation – 3 marks

Binary tree is a tree data structure (non-linear) in which each node can have at most two children which are referred to as the left child and the right child. The topmost node in a binary tree is called the root, and the bottom-most nodes are called leaves.

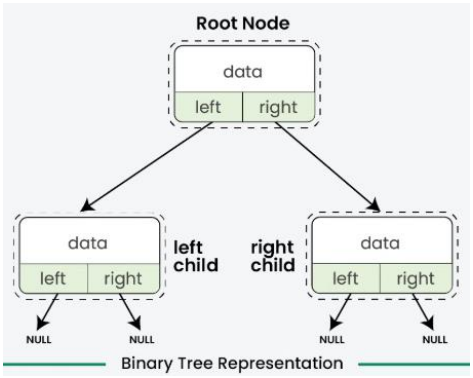
Representation of Binary Trees

There are two primary ways to represent binary trees:

- Linked Node Representation
- Array Representation

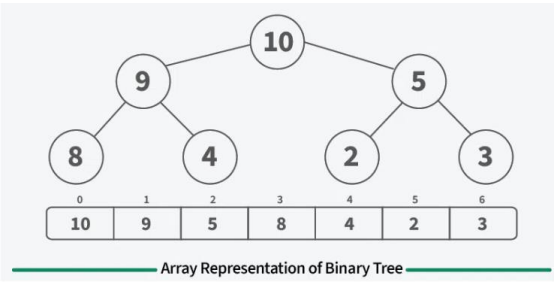
1. Linked Node Representation

This is the simplest way to represent a binary tree. Each node contains data and pointers to its left and right children.



2. Array Representation

Array Representation is another way to represent binary trees, especially useful when the tree is complete (all levels are fully filled except possibly the last, which is filled from left to right).



5b. Define Threaded Binary Trees. Discuss In-threaded Binary Tree.

A Threaded Binary Tree is a binary tree where null pointers in leaf nodes are replaced with "threads" to allow in-order traversal without the use of recursion or a stack.

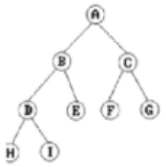
In a threaded binary tree, each node has an additional pointer called a thread, which pointsto either its in-order predecessor or successor. This allows us to efficiently traverse the tree without using recursion or a stack.

ALGORITHM:

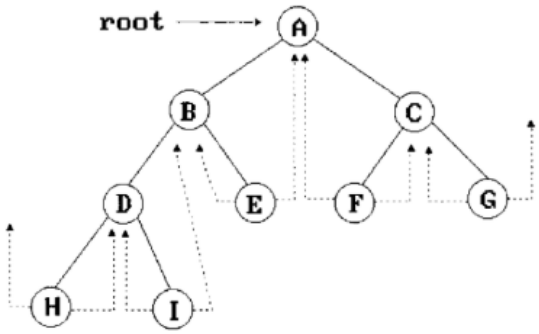
(1) If *ptr* –> *left_child* is null, replace *ptr* –> *left_child* with a pointer to the node that would be visited before *ptr* in an inorder traversal. That is we replace the null link with a pointer to the *inorder predecessor* of *ptr*.

(2) If *ptr* –> *right_child* is null, replace *ptr* –> *right_child* with a pointer to the node that would be visited after *ptr* in an inorder traversal. That is we replace the null link with a pointer to the *inorder successor* of *ptr*.

Binary tree for the following elements: A, B, C, D, E, F, G, H, I are:



Threaded Binary tree is:



5c. Write the C function for the following additional list operation:

i) Inverting Singly Linked List **ii) Concatenating Singly Linked list**

Inverting singly linked list:

```
#include <stdio.h>

struct Node {
   int data;
   struct Node* next;
};

// Given the head of a list, reverse the list and return the
// head of reversed list
struct Node* reverseList(struct Node* head) {

   // Initialize three pointers: curr, prev and next
   struct Node *curr = head, *prev = NULL, *next;

   // Traverse all the nodes of Linked List
   while (curr != NULL) {
```

```
        // Store next
        next = curr->next;

        // Reverse current node's next pointer
        curr->next = prev;

        // Move pointers one position ahead
        prev = curr;
        curr = next;
    }

    // Return the head of reversed linked list
    return prev;
}

void printList(struct Node* node) {
    while (node != NULL) {
        printf(" %d", node->data);
        node = node->next;
    }
}


struct Node* createNode(int new_data) {
    struct Node* new_node
        = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = NULL;
    return new_node;
}

int main() {

    // Create a hard-coded linked list:
    // 1 -> 2 -> 3 -> 4 -> 5
    struct Node* head = createNode(1);
    head->next = createNode(2);
    head->next->next = createNode(3);
    head->next->next->next = createNode(4);
    head->next->next->next->next = createNode(5);

    printf("Given Linked list:");
    printList(head);

    head = reverseList(head);

    printf("\nReversed Linked List:");
    printList(head);

    return 0;
}
```

Concatenating Singly Linked list:

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

struct Node{
```

```
    int data;
    struct Node *next;
};

// Function to concatenate two linked lists
struct Node *concat(struct Node *head1,
                    struct Node *head2) {

    if (head1 == NULL)
        return head2;

    // Find the last node of the first list
    struct Node *curr = head1;
    while (curr->next != NULL){
        curr = curr->next;
    }

    // Link the last node of the first list
    // to the head of the second list
    curr->next = head2;

    // Return the head of the concatenated list
    return head1;
}

void printList(struct Node *head) {
    struct Node *curr = head;
    while (curr != NULL){
        printf("%d ", curr->data);
        curr = curr->next;
    }
    printf("\n");
}

struct Node *createNode(int x) {
    struct Node *newNode =
      (struct Node *)malloc(sizeof(struct Node));
    newNode->data = x;
    newNode->next = NULL;
    return newNode;
}

int main() {

    // Create the first linked list: 1 -> 2 -> 3
    struct Node *head1 = createNode(1);
    head1->next = createNode(2);
    head1->next->next = createNode(3);

    // Create the second linked list: 4 -> 5
    struct Node *head2 = createNode(4);
    head2->next = createNode(5);

    struct Node *concatHead = concat(head1, head2);
    printList(concatHead);

    return 0;
}
```

Q.5

a) Discuss how binary trees are represented using:
i) Array
ii) Linked list

Answer:
Binary trees can be represented in two common ways: arrays and linked lists.

1. Array Representation:

- A binary tree can be stored in an array by using level-order traversal. The root node is stored at index 1 (or 0 in some implementations), and its left and right children are stored at positions 2i + 1 and 2i + 2, respectively.
- Advantages: Direct access to nodes, memory-efficient if the tree is complete.
- Disadvantages: Wastes space if the tree is sparse, resizing is difficult.

2. Linked List Representation:

- Each node in the tree is represented as an object containing data and pointers to left and right children.
- Struct definition in C:
- struct Node {
- int data;
- struct Node *left, *right;

};

- Advantages: Efficient memory usage, dynamic size.
- Disadvantages: Requires extra memory for pointers, slower access compared to arrays.

b) Define Threaded binary tree. Discuss In-threaded binary tree.

Answer:

- A **threaded binary tree** is a type of binary tree in which null pointers are replaced with special pointers called threads to improve traversal efficiency.
- **Types of Threaded Binary Trees:**
    - **Single Threaded:** Only one pointer (left or right) is replaced with a thread.
    - **Double Threaded:** Both left and right null pointers are replaced with threads.
- **In-Threaded Binary Tree:**
    - In this structure, threads replace the null left pointers to point to in-order predecessors.
    - This reduces recursion overhead and speeds up traversal.

c) Write the C function for the following additional list operations:
i) Inverting a Singly Linked List
ii) Concatenating two Singly Linked Lists

Answer:

```
struct Node {
  int data;
```

```
    struct Node *next;
};

// Function to invert a singly linked list
struct Node* invertList(struct Node* head) {
    struct Node* prev = NULL, *curr = head, *next = NULL;
    while (curr) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}

// Function to concatenate two singly linked lists
struct Node* concatenate(struct Node* list1, struct Node* list2) {
    if (!list1) return list2;
    struct Node* temp = list1;
    while (temp->next)
        temp = temp->next;
    temp->next = list2;
    return list1;
}
```

Q.6

a) Discuss Inorder, Preorder, Postorder, and Level Order Traversal with suitable functions.

Answer:

- **Inorder (Left, Root, Right):** Used in BSTs to retrieve sorted data.
- **Preorder (Root, Left, Right):** Useful in tree cloning and expression evaluation.
- **Postorder (Left, Right, Root):** Used in deletion operations and expression trees.
- **Level Order Traversal:** Uses a queue to traverse nodes level by level.

```
void inorder(struct Node* root) {
    if (root) {
        inorder(root->left);
        printf("%d ", root->data);
```

```
    inorder(root->right);
  }
}
```

b) Define Threaded Binary Tree. Construct a threaded binary tree for given elements.

Answer:

- Given elements: A, B, C, D, E, F, G, H, I
- The threaded binary tree structure is constructed with proper left and right threads.
- The implementation involves linking null pointers to their respective predecessors/successors.

c) Write a C function for the following:
i) Insert a node at the beginning of a doubly linked list.
ii) Deleting a node at the end of the doubly linked list.

```
struct DNode {
  int data;
  struct DNode *prev, *next;
};


struct DNode* insertAtBegin(struct DNode* head, int value) {
  struct DNode* newNode = (struct DNode*)malloc(sizeof(struct DNode));
  newNode->data = value;
  newNode->next = head;
  newNode->prev = NULL;
  if (head) head->prev = newNode;
  return newNode;
}


struct DNode* deleteAtEnd(struct DNode* head) {
  if (!head) return NULL;
  struct DNode* temp = head;
  while (temp->next)
    temp = temp->next;
  if (temp->prev) temp->prev->next = NULL;
  else head = NULL;
  free(temp);
  return head;
}
```

Q7 (a) Define Forest & Transform the forest into a binary tree and traverse using inorder, preorder, and postorder traversal with an example.

- Definition of Forest (2 Marks)
- Explanation of transformation from Forest to Binary Tree (3 Marks)
- Inorder, Preorder, and Postorder traversal with an example (3 Marks)

A **forest** is a collection of disjoint trees. If we remove the root of a tree with multiple subtrees, it results in a forest.

Steps to Convert a Forest into a Binary Tree:

1. Choose the leftmost node as the root.
2. Arrange each subtree by linking the first child to the left pointer and the next sibling to the right pointer.

Example: Consider a forest with three trees:

- Tree 1: A → B, C
- Tree 2: D → E, F
- Tree 3: G → H

The corresponding **binary tree** representation follows:

- Left child represents the first child.
- Right child represents the next sibling.

Traversals:

- **Inorder:** Left → Root → Right
- **Preorder:** Root → Left → Right
- **Postorder:** Left → Right → Root

Q7 (b) Define Binary search tree. Construct a binary search tree for the given elements: 100, 85, 45, 55, 120, 20, 70, 90, 115, 65, 130, 145.

- Definition of Binary Search Tree (BST) (2 Marks)
- Explanation of BST properties (2 Marks)
- Step-by-step construction of BST with given elements (2 Marks)

A **Binary Search Tree (BST)** is a binary tree where for each node:

- The left subtree contains nodes with values less than the node's value.
- The right subtree contains nodes with values greater than the node's value.

Given elements: 100, 85, 45, 55, 120, 20, 70, 90, 115, 65, 130, 145

Step-by-step Construction:

1. Start with 100 as the root.
2. Insert 85 to the left, 120 to the right.
3. Insert 45 to the left of 85, 115 to the left of 120.
4. Continue inserting elements while maintaining BST properties.

Q7 (c) Discuss Selection tree with an example.

- Definition of Selection Tree (2 Marks)
- Explanation with working and example (2 Marks)
- Diagram illustrating Selection Tree (2 Marks)

A **Selection Tree** is a complete binary tree used for **sorting** or **merging sequences** efficiently.

- **Example:** Used in tournament sorting.
- **Structure:** Leaves store input elements; internal nodes represent winners of comparisons.
- **Advantages:** Reduces comparisons and speeds up selection.

Q8 (a) Define Graph. Explain adjacency matrix and adjacency list representation with an example.

- Definition of Graph (2 Marks)
- Explanation of Adjacency Matrix representation (3 Marks)
- Explanation of Adjacency List representation (3 Marks)

A **Graph** is a data structure consisting of vertices (nodes) and edges (connections between nodes).

1. Adjacency Matrix:

- A 2D array where matrix[i][j] is 1 if there is an edge between i and j.
- **Example:**

0 1 2

0 0 1 1

1 1 0 1

2 1 1 0

2. Adjacency List:

- Uses linked lists where each node stores adjacent nodes.

- **Example:**

0 → 1 → 2

1 → 0 → 2

2 → 0 → 1

Q8 (b) Define the following terminology with example: i) Digraph ii) Weighted Graph iii) Self Loop iv) Connected Graph.

- Definition and example for Digraph (1.5 Marks)
- Definition and example for Weighted Graph (1.5 Marks)
- Definition and example for Self Loop (1.5 Marks)
- Definition and example for Connected Graph (1.5 Marks)

1. **Digraph:** A **directed graph** where edges have direction.

2. **Weighted Graph:** A graph where edges have weights (e.g., distances between cities).

3. **Self Loop:** A node with an edge to itself.

4. **Connected Graph:** Every node is reachable from any other node.

Q8 (c) Briefly explain about Elementary graph operations.

- Explanation of different elementary graph operations (3 Marks)
- Examples for each operation (3 Marks)
- 1. **Adding a vertex**
- 2. **Removing a vertex**
- 3. **Adding an edge**
- 4. **Removing an edge**
- 5. **Graph traversal (DFS, BFS)**

Q.9 Scheme

Sub Question	Marks	CO	Bloom's Level
a) Explain in detail about Static and Dynamic Hashing.	6	CO5	L2
b) What is Collision? What are the methods to resolve collision?	7	CO5	L2
c) Explain Priority queue with the help of an example.	7	CO5	L2

Solution for Q.9

(a) Static and Dynamic Hashing

Hashing is a technique used for searching and storing data efficiently. It uses a hash function to map data to a fixed-size table known as a hash table.

- **Static Hashing:**
    - The number of primary memory blocks remains fixed throughout.
    - It consists of two types: **Open Addressing** (Linear Probing, Quadratic Probing, and Double Hashing) and **Chaining** (Separate Chaining and Coalesced Chaining).
    - Insertion and deletion operations have constant time complexity, but performance degrades when the table gets filled.
- **Dynamic Hashing:**
    - The size of the hash table is adjusted dynamically based on the number of elements.
    - It uses techniques like **Extendible Hashing** and **Linear Hashing**.
    - Helps in reducing collisions and optimizing storage space.

(b) Collision and Resolution Techniques

- **Collision:**
    - Occurs when two different keys map to the same hash index.
    - Causes performance issues and must be handled efficiently.

- **Collision Resolution Techniques:**
    1. **Separate Chaining** – Each bucket maintains a linked list to store multiple values.
    2. **Open Addressing** – Finds the next available slot using probing techniques:
        - **Linear Probing** (checks the next immediate slot)
        - **Quadratic Probing** (checks farther slots using quadratic increments)
        - **Double Hashing** (uses a second hash function to resolve collisions)
    3. **Rehashing** – Creates a new larger hash table when the load factor exceeds a threshold.
    4. **Cuckoo Hashing** – Uses two hash functions and stores values in alternate locations.

(c) Priority Queue with Example

- A priority queue is a special type of queue where each element is assigned a priority.
- The element with the highest priority is dequeued first, regardless of the order in which it was enqueued.
- **Types of Priority Queues:**
    - **Max Priority Queue** – Highest value has the highest priority.
    - **Min Priority Queue** – Lowest value has the highest priority.
- **Implementation Techniques:**
    - **Array-based**
    - **Linked List-based**
    - **Binary Heap-based (Efficient)**
    - **Fibonacci Heap-based (Advanced Applications)**

Example: Consider a hospital where patients are treated based on severity. A priority queue helps in managing such cases efficiently.

```
import heapq

pq = []

heapq.heappush(pq, (1, 'Critical Patient'))

heapq.heappush(pq, (3, 'Regular Checkup'))

heapq.heappush(pq, (2, 'Emergency'))


while pq:

  print(heapq.heappop(pq))
```

Q.10 Scheme

Sub Question	Marks	CO	Bloom's Level
a) Define Hashing. Explain different hashing functions with suitable examples.	12	CO5	L2

b) Write a short note on:

i) Leftist trees

ii) Optimal binary search tree 8 CO5 L3

Solution for Q.10

(a) Hashing and Hashing Functions

- **Definition of Hashing:**
  - Hashing is a technique that converts large keys into smaller ones using a hash function.
  - Used in database indexing, caching, and search optimization.
- **Types of Hash Functions:**
  1. **Division Method** – h(k) = k mod m
  2. **Multiplication Method** – h(k) = floor(m * (k * A mod 1))
  3. **Folding Method** – Breaks key into parts and sums them.
  4. **Mid-Square Method** – Squares the key and extracts the middle portion.
  5. **Universal Hashing** – Uses a randomly selected hash function from a set.
- **Example:**

```
class HashTable:
    def __init__(self, size):
        self.size = size
        self.table = [None] * size

    def insert(self, key, value):
        index = key % self.size
        self.table[index] = value

    def display(self):
        print(self.table)

ht = HashTable(10)
ht.insert(23, 'Data1')
ht.insert(56, 'Data2')
ht.display()
```

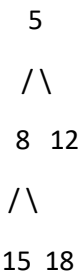
(b) Leftist Trees and Optimal Binary Search Trees

(i) Leftist Trees

- A **leftist tree** is a variant of a **binary heap** that favors merging operations.

- Maintains a **null-path length** (NPL), ensuring left child has a greater or equal NPL than the right child.
- **Applications:** Efficient priority queues, dynamic merging of heaps.

Example:



(ii) Optimal Binary Search Tree (OBST)

- A BST that minimizes the expected search cost.
- Uses **dynamic programming** to determine the optimal structure based on access probabilities.
- **Formula:**

Cost(i, j) = min [Cost(i, k) + Cost(k+1, j) + sum(freq[i:j])]

- **Example:** Used in compiler design, database indexing.