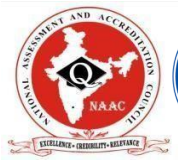
QMP 7.1 D/F



Channabasaveshwara Institute of Technology

(Affiliated to VTU, Belgavi & Approved by AICTE, New Delhi)
(NAAC Accredited & ISO 9001:2015 Certified Institution)
NH 206 (B.H. Road), Gubbi, Tumkur – 572 216. Karnataka



DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

DATA STRUCTURES LABORATORY

&

1BCSL306

B.E - III Semester Lab Manual

Academic Year: 2026-27

Name : _____

USN : _____

Batch :_____Section : _____

Mail -ID :_____Contact Number:

QMP 7.1 D/F

Channabasaveshwara Institute of Technology

(Affiliated to VTU, Belgavi & Approved by AICTE, New Delhi)
(NAAC Accredited & ISO 9001:2015 Certified Institution)
NH 206 (B.H. Road), Gubbi, Tumkur – 572 216. Karnataka

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

DATA STRUCTURES LABORATORY

&

1BCSL306

B.E – III Semester Lab Manual

Prepared By
Mr. Anil Kumar
Asst. Prof., Dept. of ISE

Reviewed by:
Mr. Pradeep M
Asst. Prof., Dept. of ISE

Approved by:

Dr. Thara D K
Professor & HOD,
Dept. of ISE, CIT Gubbi

Channabasaveshwara Institute of Technology
(Affiliated to VTU, Belgavi & Approved by AICTE, New Delhi)
(NAAC Accredited & ISO 9001:2015 Certified Institution)
NH 206 (B.H. Road), Gubbi, Tumkur – 572 216. Karnataka

INSTITUTE VISION

To create centers of excellence in education and to serve the society by enhancing the quality of life through value based professional leadership

INSTITUTE MISSION

- To provide high quality technical and professionally relevant education in a diverse learning environment
- To provide the values that prepare students to lead their lives with personal integrity, professional ethics and civic responsibility in a global society
- To prepare the next generation of skilled professionals to successfully compete in the diverse global market
- To promote a campus environment that welcomes and honors women and men of all races, creeds and cultures, values and intellectual curiosity, pursuit of knowledge and academic integrity and freedom
- To offer a wide variety of off-campus education and training programmes to individuals and groups
- To stimulate collaborative efforts with industry, universities, government and professional societies
- To facilitate public understanding of technical issues and achieve excellence in the operations of the institute

QUALITY POLICY

Our organization delights customers (students, parents and society) by providing value added quality education to meet the national and international requirements. We also provide necessary steps to train the students for placement and continue to improve our methods of education to the students through effective quality management system, quality policy and quality objectives.

Channabasaveshwara Institute of Technology

(Affiliated to VTU, Belgavi & Approved by AICTE, New Delhi)

(NAAC Accredited & ISO 9001:2015 Certified Institution)

NH 206 (B.H. Road), Gubbi, Tumkur – 572 216. Karnataka

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

Vision

To educate and train students by imparting strong foundational knowledge in Information Science and Engineering, while creating centres of excellence in computing education

Mission

- By providing qualified and dedicated human resources, we ensure students receive the best education in Information Science and Engineering
- By providing the state-of-the-art infrastructure, we facilitate a conducive learning environment for students pursuing excellence in the field
- By fostering collaboration with industry and research institutions, we enhance students' exposure to practical applications and cutting-edge developments
- By instilling a sense of social responsibility and promoting professional ethics, we prepare students to be ethical and socially conscious professionals in their careers.

Program Outcomes (POs)

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Engineering Tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and the world**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
8. **Individual and Collaborative team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
9. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
10. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
11. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Educational Objectives (PEOs)

After four Years of Graduation, our graduates can:

PEO1: Develop the next generation of highly skilled graduates equipped with a strong knowledge in Information Science and Engineering for creating innovative solutions to society's pressing challenges.

PEO2: To motivate students to pursue higher studies in various disciplines in Information Science and Engineering with the aim of cherishing careers in research and development, academia, industry and entrepreneurship.

PEO3: To Produce engineers who are professionals' entrepreneurs and capable of self-learning to excel in their career.

PEO4: To prepare graduates who excel in diverse learns upholding professional ethics and societal responsibilities.

Program Specific Outcomes (PSOs)

At the time of graduation, our graduates can:

PSO1: Students will possess the ability to analyze, design, develop, test and apply mathematical foundations, management principles, and adapt emerging technologies effectively in the creation of computational solutions.

PSO2: Students will possess advanced skills in artificial intelligence, cybersecurity, cloud computing, project management, mobile application development, embedded systems, and big data analytics, making them highly employable, well prepared for higher education, and capable of contributing to cutting-edge research and innovation in Information Science and Engineering.

Instructions to the Candidates

- Student should come with thorough preparation for the experiment to be conducted.
- Student should take prior permission from the concerned faculty before availing the leave.
- Student should come with proper dress code and to be present on time in the laboratory.
- Student will not be permitted to attend the laboratory unless they bring the practical record fully completed in all respects pertaining to the experiment conducted in the previous class.
- Student will not be permitted to attend the laboratory unless they bring the observation book fully completed in all respects pertaining to the experiment to be conducted in present class.
- Practical record and observation book should be neatly maintained.
- Student should obtain the signature of the staff-in-charge in the observation book after completing each experiment.
- Theory related to each experiment should be written in the practical record before procedure in your own words with appropriate references.

CONTENTS

Part-A: Fixed Experiments

1. Library Book Management System (Structures, Dynamic Memory)

2. Stack Operations using Arrays (Push, Pop, Palindrome Check)

3. Printer Queue Simulation using Linear Queue

4. Singly Linked List of Student Records

5. Binary Tree — Level-Order Creation and Traversals

6. Graph Representation and Traversal (DFS/BFS)

Part-B: Open-Ended Experiments

1. Sparse Matrix Addition using 3-Tuple Representation

2. Infix to Postfix Conversion using Stack

3. Circular Queue Operations

4. Doubly Linked List of Employee Records (with DEQ)

5. Binary Search Tree Operations

6. Employee Record Management using Hashing (Linear Probing)

COURSE OUTCOMES	
CO1:	Develop programs to implement fundamental and linear data structures and their operations for solving computational problems.
CO2:	Develop programs using non-linear data structures and hashing techniques to solve computational problems.
CO3:	Select appropriate data structure to organize, process, and retrieve data while solving real-world problems.
CO4:	Build experimental documents involving data structures, algorithms, program design, implementation and results with inferences.

RBT LEVELS						
COS	**Remember (L1)**	**Understand (L2)**	**Apply (L3)**	**Analyze (L4)**	**Evaluate (L5)**	**Create (L6)**
CO1			√			
CO2			√			
CO3					√	
CO4						√

CO – PO MAPPING														
CO-PO	**PO1**	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**	**PSO1**	**PSO2**
CO1	3	3	3	2	2				1	1			2	2
CO2	3	3	3	2	2				1	1			2	2
CO3	3	3	3	3	2				2	2			2	2
CO4	2	2	2	3	2				1	2			2	2
Average	**2.75**	**2.75**	**2.75**	**2.5**	**2**				**1.25**	**1.5**			**2**	**2**

DATA STRUCTURES LABORATORY

Solutions – Part A: Fixed Experiments & Part B: Open-Ended Experiments

Language: C | Suitable for laboratory record and practical examination

General Notes for Laboratory Execution

• Compile with a C compiler such as GCC: gcc program.c -o program

• Run: ./program (Linux/macOS) or program.exe (Windows).

• For pointer-based programs, always check dynamic allocation and free allocated nodes when appropriate.

• For the sparse matrix addition program, enter non-zero terms in row-major order.

• In the infix-to-postfix program, operands are treated as single characters; expressions such as (A+B)*C-D/E are supported.

• The examples are intentionally straightforward so that the logic can be explained during a practical examination.

Experiment 1: Library Book Management System (Structures, Dynamic Memory)

Aim

To implement a library book management system using structures and dynamically allocated memory, supporting insertion, display and search of books.

Algorithm / Procedure

1. Define a Book structure containing book ID, title, author and availability status.

2. Allocate memory for n book records using malloc().

3. Read book details into the allocated array.

4. Display all records.

5. Search for a book using its ID and display the matching record.

6. Release dynamically allocated memory using free().

C Program

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

typedef struct {
    int id;
    char title[50];
    char author[50];
    int available;
} Book;

int main(void) {
```

```
    int n, i, key, found = 0;
    Book *b;

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

    b = (Book *)malloc(n * sizeof(Book));
    if (b == NULL) {
        printf("Memory allocation failed.\\n");
        return 1;
    }

    for (i = 0; i < n; i++) {
        printf("\\nBook %d ID: ", i + 1);
        scanf("%d", &b[i].id);
        printf("Title: ");
        scanf(" %49[^\\n]", b[i].title);
        printf("Author: ");
        scanf(" %49[^\\n]", b[i].author);
        printf("Available? (1-Yes, 0-No): ");
        scanf("%d", &b[i].available);
    }

    printf("\\n--- Library Records ---\\n");
    for (i = 0; i < n; i++)
        printf("%d | %s | %s | %s\\n", b[i].id, b[i].title,
               b[i].author, b[i].available ? "Available" : "Issued");

    printf("\\nEnter Book ID to search: ");
    scanf("%d", &key);

    for (i = 0; i < n; i++) {
        if (b[i].id == key) {
            printf("Found: %d | %s | %s | %s\\n", b[i].id, b[i].title,
                   b[i].author, b[i].available ? "Available" : "Issued");
            found = 1;
            break;
        }
    }
    if (!found)
        printf("Book not found.\\n");

    free(b);
    return 0;
}
```

Sample Output

```
Enter number of books: 2
Book 1 ID: 101
Title: Data Structures
Author: Mark Allen
Available? (1-Yes, 0-No): 1
Book 2 ID: 102
Title: C Programming
Author: Dennis Ritchie
Available? (1-Yes, 0-No): 0
```

```
--- Library Records ---
101 | Data Structures | Mark Allen | Available
102 | C Programming | Dennis Ritchie | Issued

Enter Book ID to search: 102
Found: 102 | C Programming | Dennis Ritchie | Issued
```

Complexity

Display/search: O(n) time. Dynamic storage: O(n) space.

Experiment 2: Stack Operations using Arrays (Push, Pop, Palindrome Check)

Aim

To implement stack operations using an array and use the stack to check whether a string is a palindrome.

Algorithm / Procedure

1. Initialize top = -1.

2. Push an item by incrementing top and storing the item.

3. Pop an item by returning stack[top] and decrementing top.

4. For palindrome checking, push every character of the string and compare the popped characters with the original string.

5. Report whether the string is a palindrome.

C Program

```
#include <stdio.h>
#include <string.h>
#define MAX 100

char stack[MAX];
int top = -1;

void push(char x) {
    if (top == MAX - 1) printf("Stack Overflow\\n");
    else stack[++top] = x;
}

char pop(void) {
    if (top == -1) return '\0';
    return stack[top--];
}

int main(void) {
    char s[MAX];
    int i, palindrome = 1;

    printf("Enter a string: ");
    scanf(" %99[^\\n]", s);

    for (i = 0; s[i] != '\0'; i++)
        push(s[i]);

    for (i = 0; s[i] != '\0'; i++) {
        if (s[i] != pop()) {
            palindrome = 0;
            break;
        }
    }

    printf("Original string: %s\\n", s);
    if (palindrome) printf("Palindrome\\n");
    else printf("Not a palindrome\\n");
```

```
    return 0;
}
```

Sample Output

```
Enter a string: MADAM
Original string: MADAM
Palindrome
```

Complexity

Push/pop: O(1) each. Palindrome check: O(n) time and O(n) auxiliary space.

Experiment 3: Printer Queue Simulation using Linear Queue

Aim

To simulate printer jobs using a linear queue implemented with an array.

Algorithm / Procedure

1. Initialize front = rear = -1.

2. Enqueue a print job at rear.

3. Dequeue the job at front and process it.

4. Display the jobs currently waiting.

5. Because this is a linear queue, when rear reaches MAX-1 no further insertion is possible even if positions before front are empty.

C Program

```
#include <stdio.h>
#define MAX 5

int q[MAX], front = -1, rear = -1;

void enqueue(int job) {
    if (rear == MAX - 1) {
        printf("Queue Overflow\\n");
        return;
    }
    if (front == -1) front = 0;
    q[++rear] = job;
    printf("Job %d added.\\n", job);
}

void dequeue(void) {
    if (front == -1 || front > rear) {
        printf("Queue Underflow\\n");
        return;
    }
    printf("Printing job %d\\n", q[front++]);
    if (front > rear) front = rear = -1;
}

void display(void) {
    int i;
    if (front == -1) {
        printf("Queue is empty.\\n");
        return;
    }
    printf("Waiting jobs: ");
    for (i = front; i <= rear; i++) printf("%d ", q[i]);
    printf("\\n");
}

int main(void) {
    enqueue(101);
    enqueue(102);
    enqueue(103);
```

```
    display();
    dequeue();
    display();
    enqueue(104);
    display();
    return 0;
}
```

Sample Output

```
Job 101 added.
Job 102 added.
Job 103 added.
Waiting jobs: 101 102 103
Printing job 101
Waiting jobs: 102 103
Job 104 added.
Waiting jobs: 102 103 104
```

Complexity

Enqueue/dequeue: O(1). Display: O(n). Space: O(n).

Experiment 4: Singly Linked List of Student Records

Aim

To create and manipulate a singly linked list containing student records.

Algorithm / Procedure

1. Define a node containing USN, name, marks and a next pointer.

2. Create nodes dynamically using malloc().

3. Insert each student at the end of the list.

4. Traverse the list and display all student records.

5. Free every node before program termination.

C Program

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

typedef struct Node {
    char usn[20], name[50];
    float marks;
    struct Node *next;
} Node;

Node *insertEnd(Node *head) {
    Node *p = (Node *)malloc(sizeof(Node));
    Node *q;
    if (!p) exit(1);

    printf("USN: "); scanf("%19s", p->usn);
    printf("Name: "); scanf(" %49[^\\n]", p->name);
    printf("Marks: "); scanf("%f", &p->marks);
    p->next = NULL;

    if (!head) return p;
    q = head;
    while (q->next) q = q->next;
    q->next = p;
    return head;
}

void display(Node *head) {
    if (!head) {
        printf("List is empty.\\n");
        return;
    }
    while (head) {
        printf("%s | %s | %.2f\\n", head->usn, head->name, head->marks);
        head = head->next;
    }
}

void freeList(Node *head) {
    Node *t;
    while (head) {
```

```
            t = head;
            head = head->next;
            free(t);
        }
    }

    int main(void) {
        Node *head = NULL;
        int n, i;
        printf("Enter number of students: ");
        scanf("%d", &n);

        for (i = 0; i < n; i++) {
            printf("\\nStudent %d\\n", i + 1);
            head = insertEnd(head);
        }

        printf("\\nStudent Records\\n");
        display(head);
        freeList(head);
        return 0;
    }
```

Sample Output

```
Enter number of students: 2
Student 1
USN: 1CI23IS001
Name: Anu
Marks: 86
Student 2
USN: 1CI23IS002
Name: Ravi
Marks: 91

Student Records
1CI23IS001 | Anu | 86.00
1CI23IS002 | Ravi | 91.00
```

Complexity

Traversal: O(n). Insertion at end as written: O(n) per insertion; $O(n^2)$ for n insertions. Space: O(n).

Experiment 5: Binary Tree — Level-Order Creation and Traversals

Aim

To construct a binary tree from level-order input and perform preorder, inorder, postorder and level-order traversals.

Algorithm / Procedure

1. Read level-order values, using -1 to represent a missing child.

2. Create the root and use a queue of node pointers.

3. For each dequeued node, read/create its left and right children.

4. Perform recursive preorder, inorder and postorder traversals.

5. Perform level-order traversal using a queue.

C Program

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100

typedef struct Node {
    int data;
    struct Node *left, *right;
} Node;

Node *newNode(int x) {
    Node *p = (Node *)malloc(sizeof(Node));
    p->data = x; p->left = p->right = NULL;
    return p;
}

Node *createLevelOrder(int a[], int n) {
    Node *q[MAX];
    int front = 0, rear = 0, i = 0;
    Node *root;

    if (n == 0 || a[0] == -1) return NULL;
    root = newNode(a[0]);
    q[rear++] = root;

    while (front < rear && i < n) {
        Node *cur = q[front++];
        if (++i < n && a[i] != -1) {
            cur->left = newNode(a[i]);
            q[rear++] = cur->left;
        }
        if (++i < n && a[i] != -1) {
            cur->right = newNode(a[i]);
            q[rear++] = cur->right;
        }
    }
    return root;
}

void preorder(Node *r) {
    if (r) { printf("%d ", r->data); preorder(r->left); preorder(r->right); }
```

```
}
void inorder(Node *r) {
    if (r) { inorder(r->left); printf("%d ", r->data); inorder(r->right); }
}
void postorder(Node *r) {
    if (r) { postorder(r->left); postorder(r->right); printf("%d ", r->data); }
}
void levelorder(Node *r) {
    Node *q[MAX]; int f = 0, rr = 0;
    if (!r) return;
    q[rr++] = r;
    while (f < rr) {
        Node *p = q[f++];
        printf("%d ", p->data);
        if (p->left) q[rr++] = p->left;
        if (p->right) q[rr++] = p->right;
    }
}

int main(void) {
    int a[MAX], n, i;
    Node *root;

    printf("Enter number of level-order positions: ");
    scanf("%d", &n);
    printf("Enter values (-1 for NULL): ");
    for (i = 0; i < n; i++) scanf("%d", &a[i]);

    root = createLevelOrder(a, n);

    printf("Preorder: "); preorder(root);
    printf("\\nInorder: "); inorder(root);
    printf("\\nPostorder: "); postorder(root);
    printf("\\nLevel order: "); levelorder(root);
    printf("\\n");
    return 0;
}
```

Sample Output

```
Enter number of level-order positions: 7
Enter values (-1 for NULL): 1 2 3 4 5 6 7
Preorder: 1 2 4 5 3 6 7
Inorder: 4 2 5 1 6 3 7
Postorder: 4 5 2 6 7 3 1
Level order: 1 2 3 4 5 6 7
```

Complexity

Tree construction and level-order traversal: O(n). Each depth-first traversal: O(n). Auxiliary queue/recursion: O(n) worst case.

Experiment 6: Graph Representation and Traversal (DFS/BFS)

Aim

To represent a graph using an adjacency matrix and perform Depth First Search (DFS) and Breadth First Search (BFS).

Algorithm / Procedure

1. Read the number of vertices and the adjacency matrix.

2. Choose a starting vertex.

3. DFS recursively visits an unvisited vertex and then its adjacent vertices.

4. BFS uses a queue to visit vertices level by level.

5. Maintain a visited array for both traversals.

C Program

```
#include <stdio.h>
#define MAX 20

int g[MAX][MAX], visited[MAX], n;

void dfs(int v) {
    int i;
    visited[v] = 1;
    printf("%d ", v);
    for (i = 0; i < n; i++)
        if (g[v][i] && !visited[i])
            dfs(i);
}

void bfs(int start) {
    int q[MAX], front = 0, rear = 0, i;
    for (i = 0; i < n; i++) visited[i] = 0;

    q[rear++] = start;
    visited[start] = 1;

    while (front < rear) {
        int v = q[front++];
        printf("%d ", v);
        for (i = 0; i < n; i++) {
            if (g[v][i] && !visited[i]) {
                visited[i] = 1;
                q[rear++] = i;
            }
        }
    }
}

int main(void) {
    int i, j, start;
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter adjacency matrix:\\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
```

```
            scanf("%d", &g[i][j]);

    printf("Start vertex (0 to %d): ", n - 1);
    scanf("%d", &start);

    for (i = 0; i < n; i++) visited[i] = 0;
    printf("DFS: "); dfs(start);

    printf("\\nBFS: "); bfs(start);
    printf("\\n");
    return 0;
}
```

Sample Output

```
Enter number of vertices: 5
Enter adjacency matrix:
0 1 1 0 0
1 0 0 1 0
1 0 0 0 1
0 1 0 0 1
0 0 1 1 0
Start vertex (0 to 4): 0
DFS: 0 1 3 4 2
BFS: 0 1 2 3 4
```

Complexity

With adjacency matrix, DFS/BFS take $O(V^2)$ time. Space: $O(V^2)$ for matrix plus O(V) traversal storage.

Part-B: Open-Ended Experiments

Experiment 1: Sparse Matrix Addition using 3-Tuple Representation

Aim

To represent sparse matrices using 3-tuples and add two sparse matrices efficiently.

Algorithm / Procedure

1. Store each non-zero element as (row, column, value).

2. Store the first tuple entry as (number of rows, number of columns, number of non-zero elements).

3. Compare positions of non-zero elements in the two tuples.

4. Copy the smaller position, or add values when positions are equal.

5. Display the resulting sparse matrix in 3-tuple form.

C Program

```
#include <stdio.h>
#define MAX 100

typedef struct {
    int r, c, v;
} Term;

void readSparse(Term a[]) {
    int rows, cols, n, i;
    printf("Rows, columns and non-zero count: ");
    scanf("%d%d%d", &rows, &cols, &n);
    a[0] = (Term){rows, cols, n};
    printf("Enter row column value for each non-zero term:\\n");
    for (i = 1; i <= n; i++)
        scanf("%d%d%d", &a[i].r, &a[i].c, &a[i].v);
}

int addSparse(Term a[], Term b[], Term s[]) {
    int i = 1, j = 1, k = 1;
    s[0].r = a[0].r; s[0].c = a[0].c;

    while (i <= a[0].v && j <= b[0].v) {
        if (a[i].r == b[j].r && a[i].c == b[j].c) {
            int sum = a[i].v + b[j].v;
            if (sum != 0) s[k++] = (Term){a[i].r, a[i].c, sum};
            i++; j++;
        } else if (a[i].r < b[j].r ||
                  (a[i].r == b[j].r && a[i].c < b[j].c))
            s[k++] = a[i++];
        else
            s[k++] = b[j++];
    }
    while (i <= a[0].v) s[k++] = a[i++];
    while (j <= b[0].v) s[k++] = b[j++];

    s[0].v = k - 1;
```

```
    return k;
}

void display(Term a[]) {
    int i;
    printf("Row Col Value\\n");
    for (i = 0; i <= a[0].v; i++)
        printf("%d   %d   %d\\n", a[i].r, a[i].c, a[i].v);
}

int main(void) {
    Term a[MAX], b[MAX], s[MAX];
    readSparse(a);
    readSparse(b);

    if (a[0].r != b[0].r || a[0].c != b[0].c) {
        printf("Matrix dimensions do not match.\\n");
        return 0;
    }

    addSparse(a, b, s);
    printf("\\nResult in 3-tuple form:\\n");
    display(s);
    return 0;
}
```

Sample Output

```
Rows, columns and non-zero count: 3 3 2
Enter row column value for each non-zero term:
0 0 5
1 2 7
Rows, columns and non-zero count: 3 3 2
Enter row column value for each non-zero term:
0 0 3
2 1 4

Result in 3-tuple form:
Row Col Value
3   3   3
0   0   8
1   2   7
2   1   4
```

Complexity

If terms are stored in sorted row-column order, addition is O(t1+t2), where t1 and t2 are the non-zero counts. Space: O(t1+t2).

Experiment 2: Infix to Postfix Conversion using Stack

Aim

To convert an infix arithmetic expression to postfix notation using an operator stack.

Algorithm / Procedure

1. Scan the expression from left to right.

2. Output operands immediately.

3. Push '(' onto the stack.

4. For ')', pop operators until '(' is found.

5. For an operator, pop operators having higher or equal precedence (except '('), then push the current operator.

6. After scanning, pop all remaining operators.

C Program

```
#include <stdio.h>
#include <ctype.h>
#define MAX 100

char st[MAX];
int top = -1;

int prec(char c) {
    if (c == '^') return 3;
    if (c == '*' || c == '/' || c == '%') return 2;
    if (c == '+' || c == '-') return 1;
    return 0;
}

int main(void) {
    char infix[MAX], postfix[MAX];
    int i = 0, k = 0;
    char c;

    printf("Enter infix expression: ");
    scanf("%99s", infix);

    while ((c = infix[i++]) != '\0') {
        if (isalnum((unsigned char)c))
            postfix[k++] = c;
        else if (c == '(')
            st[++top] = c;
        else if (c == ')') {
            while (top >= 0 && st[top] != '(')
                postfix[k++] = st[top--];
            if (top >= 0) top--;
        } else {
            while (top >= 0 && st[top] != '(' &&
                   prec(st[top]) >= prec(c))
                postfix[k++] = st[top--];
            st[++top] = c;
        }
    }
```

```
    while (top >= 0)
        postfix[k++] = st[top--];
    postfix[k] = '\0';

    printf("Postfix: %s\\n", postfix);
    return 0;
}
```

Sample Output

```
Enter infix expression: (A+B)*C-D/E
Postfix: AB+C*DE/-
```

Complexity

Time: O(n). Auxiliary stack space: O(n). Note: this version assumes single-character operands and standard binary operators.

Experiment 3: Circular Queue Operations

Aim

To implement enqueue, dequeue and display operations in a circular queue using an array.

Algorithm / Procedure

1. Initialize front = rear = -1.

2. Queue is full when the next rear position equals front.

3. On insertion, move rear circularly using (rear+1)%MAX.

4. On deletion, move front circularly using (front+1)%MAX.

5. Reset both indices to -1 when the last item is deleted.

C Program

```
#include <stdio.h>
#define MAX 5

int q[MAX], front = -1, rear = -1;

int isEmpty(void) { return front == -1; }
int isFull(void) { return !isEmpty() && (rear + 1) % MAX == front; }

void enqueue(int x) {
    if (isFull()) {
        printf("Circular queue overflow\\n");
        return;
    }
    if (isEmpty()) front = rear = 0;
    else rear = (rear + 1) % MAX;
    q[rear] = x;
}

void dequeue(void) {
    if (isEmpty()) {
        printf("Circular queue underflow\\n");
        return;
    }
    printf("Deleted: %d\\n", q[front]);
    if (front == rear) front = rear = -1;
    else front = (front + 1) % MAX;
}

void display(void) {
    int i;
    if (isEmpty()) {
        printf("Queue is empty\\n");
        return;
    }
    printf("Queue: ");
    i = front;
    while (1) {
        printf("%d ", q[i]);
        if (i == rear) break;
        i = (i + 1) % MAX;
```

```
    }
    printf("\\n");
}

int main(void) {
    enqueue(10); enqueue(20); enqueue(30); enqueue(40);
    display();
    dequeue(); dequeue();
    enqueue(50); enqueue(60);
    display();
    return 0;
}
```

Sample Output

```
Queue: 10 20 30 40
Deleted: 10
Deleted: 20
Queue: 30 40 50 60
```

Complexity

Enqueue/dequeue: O(1). Display: O(n). Space: O(n).

Experiment 4: Doubly Linked List of Employee Records (with DEQ)

Aim

To implement a doubly linked list of employee records and use it as a double-ended queue (DEQ), supporting insertion/deletion at both ends.

Algorithm / Procedure

1. Define a doubly linked node with employee details and prev/next pointers.

2. Insert an employee at the front or rear.

3. Delete an employee from the front or rear.

4. Display employees from front to rear.

5. Maintain both front and rear pointers.

C Program

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

typedef struct Node {
    int id;
    char name[40];
    float salary;
    struct Node *prev, *next;
} Node;

Node *front = NULL, *rear = NULL;

Node *makeNode(void) {
    Node *p = (Node *)malloc(sizeof(Node));
    printf("Employee ID: "); scanf("%d", &p->id);
    printf("Name: "); scanf(" %39[^\\n]", p->name);
    printf("Salary: "); scanf("%f", &p->salary);
    p->prev = p->next = NULL;
    return p;
}

void insertFront(void) {
    Node *p = makeNode();
    if (!front) front = rear = p;
    else {
        p->next = front; front->prev = p; front = p;
    }
}

void insertRear(void) {
    Node *p = makeNode();
    if (!rear) front = rear = p;
    else {
        p->prev = rear; rear->next = p; rear = p;
    }
}

void deleteFront(void) {
    Node *p;
```

```
    if (!front) { printf("DEQ is empty\\n"); return; }
    p = front;
    printf("Deleted employee: %d\\n", p->id);
    front = front->next;
    if (front) front->prev = NULL; else rear = NULL;
    free(p);
}

void deleteRear(void) {
    Node *p;
    if (!rear) { printf("DEQ is empty\\n"); return; }
    p = rear;
    printf("Deleted employee: %d\\n", p->id);
    rear = rear->prev;
    if (rear) rear->next = NULL; else front = NULL;
    free(p);
}

void display(void) {
    Node *p = front;
    while (p) {
        printf("%d | %s | %.2f\\n", p->id, p->name, p->salary);
        p = p->next;
    }
}

int main(void) {
    printf("Insert at rear:\\n"); insertRear();
    printf("Insert at rear:\\n"); insertRear();
    printf("Insert at front:\\n"); insertFront();

    printf("\\nDEQ contents:\\n");
    display();

    deleteFront();
    deleteRear();

    printf("\\nAfter deletions:\\n");
    display();
    return 0;
}
```

Sample Output

```
Insert at rear:
Employee ID: 101
Name: Ravi
Salary: 45000
Insert at rear:
Employee ID: 102
Name: Anu
Salary: 50000
Insert at front:
Employee ID: 103
Name: Kiran
Salary: 48000

DEQ contents:
103 | Kiran | 48000.00
101 | Ravi | 45000.00
```

```
102 | Anu | 50000.00
Deleted employee: 103
Deleted employee: 102

After deletions:
101 | Ravi | 45000.00
```

Complexity

Insert/delete at either end: O(1). Display: O(n). Space: O(n).

Experiment 5: Binary Search Tree Operations

Aim

To implement insertion, search, inorder, preorder and postorder operations on a Binary Search Tree (BST).

Algorithm / Procedure

1. Create a BST node dynamically.

2. Insert a key: smaller values go to the left subtree and larger values to the right.

3. Search recursively according to the BST property.

4. Use inorder traversal to obtain keys in sorted order.

5. Use preorder and postorder to demonstrate other traversals.

C Program

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

typedef struct Node {
    int key;
    struct Node *left, *right;
} Node;

Node *newNode(int key) {
    Node *p = (Node *)malloc(sizeof(Node));
    p->key = key; p->left = p->right = NULL;
    return p;
}

Node *insert(Node *root, int key) {
    if (!root) return newNode(key);
    if (key < root->key) root->left = insert(root->left, key);
    else if (key > root->key) root->right = insert(root->right, key);
    return root;
}

Node *search(Node *root, int key) {
    if (!root || root->key == key) return root;
    if (key < root->key) return search(root->left, key);
    return search(root->right, key);
}

void inorder(Node *r) {
    if (r) { inorder(r->left); printf("%d ", r->key); inorder(r->right); }
}
void preorder(Node *r) {
    if (r) { printf("%d ", r->key); preorder(r->left); preorder(r->right); }
}
void postorder(Node *r) {
    if (r) { postorder(r->left); postorder(r->right); printf("%d ", r->key); }
}

int main(void) {
    Node *root = NULL;
    int keys[] = {50, 30, 70, 20, 40, 60, 80};
```

```
    int i, key;

    for (i = 0; i < 7; i++) root = insert(root, keys[i]);

    printf("Inorder: "); inorder(root);
    printf("\\nPreorder: "); preorder(root);
    printf("\\nPostorder: "); postorder(root);

    printf("\\nEnter key to search: ");
    scanf("%d", &key);
    printf(search(root, key) ? "Key found\\n" : "Key not found\\n");
    return 0;
}
```

Sample Output

```
Inorder: 20 30 40 50 60 70 80
Preorder: 50 30 20 40 70 60 80
Postorder: 20 40 30 60 80 70 50
Enter key to search: 60
Key found
```

Complexity

Average insertion/search: O(log n); worst case for a skewed BST: O(n). Traversals: O(n). Space: O(n) for nodes.

Experiment 6: Employee Record Management using Hashing (Linear Probing)

Aim

To implement employee record storage and search using a hash table with linear probing for collision resolution.

Algorithm / Procedure

1. Create a hash table initialized to EMPTY.

2. Use hash(key) = key % TABLE_SIZE.

3. On collision, check successive positions using (index+1)%TABLE_SIZE.

4. Insert the record into the first available slot.

5. Search using the same probing sequence until the key is found or an unused slot is reached.

6. Display the hash table.

C Program

```
#include <stdio.h>
#include <string.h>
#define SIZE 10
#define EMPTY -1

typedef struct {
    int id;
    char name[40];
    float salary;
} Employee;

Employee table[SIZE];

void init(void) {
    int i;
    for (i = 0; i < SIZE; i++) table[i].id = EMPTY;
}

void insert(Employee e) {
    int start = e.id % SIZE, i, pos;
    for (i = 0; i < SIZE; i++) {
        pos = (start + i) % SIZE;
        if (table[pos].id == EMPTY) {
            table[pos] = e;
            printf("Inserted at index %d\\n", pos);
            return;
        }
    }
    printf("Hash table is full\\n");
}

int search(int id) {
    int start = id % SIZE, i, pos;
    for (i = 0; i < SIZE; i++) {
        pos = (start + i) % SIZE;
        if (table[pos].id == EMPTY) return -1;
        if (table[pos].id == id) return pos;
    }
```

```
    return -1;
}

void display(void) {
    int i;
    printf("\\nIndex | ID | Name | Salary\\n");
    for (i = 0; i < SIZE; i++) {
        if (table[i].id == EMPTY)
            printf("%d | ---\\n", i);
        else
            printf("%d | %d | %s | %.2f\\n", i, table[i].id,
                   table[i].name, table[i].salary);
    }
}

int main(void) {
    Employee e;
    int n, i, key, pos;

    init();
    printf("Number of employees: ");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        printf("\\nEmployee %d ID: ", i + 1);
        scanf("%d", &e.id);
        printf("Name: ");
        scanf(" %39[^\\n]", e.name);
        printf("Salary: ");
        scanf("%f", &e.salary);
        insert(e);
    }

    display();

    printf("\\nEnter employee ID to search: ");
    scanf("%d", &key);
    pos = search(key);
    if (pos == -1) printf("Employee not found\\n");
    else printf("Found at index %d: %s, %.2f\\n",
                pos, table[pos].name, table[pos].salary);

    return 0;
}
```

Sample Output

```
Number of employees: 3
Employee 1 ID: 101
Name: Ravi
Salary: 45000
Inserted at index 1

Employee 2 ID: 111
Name: Anu
Salary: 50000
Inserted at index 2

Employee 3 ID: 121
Name: Kiran
```

```
Salary: 55000
Inserted at index 3

Enter employee ID to search: 111
Found at index 2: Anu, 50000.00
```

Complexity

Average insertion/search: O(1) with a good load factor; worst case: O(n). Table space: O(m), where m is table size.

Part-A: Fixed Experiments

1. Library Book Management System – Structures & Dynamic Memory

1. **What is a structure in C?**
   A structure is a user-defined data type that groups variables of different data types under one name.

2. **Why is dynamic memory allocation used in this experiment?**
   It allows memory for book records to be allocated at runtime according to the number of books required.

3. **Which functions are used for dynamic memory allocation?**
   malloc(), calloc(), realloc() and free().

4. **What is the purpose of free()?**
   free() releases dynamically allocated memory back to the system.

5. **What is the advantage of using a structure for book records?**
   All information related to a book, such as ID, title, author and availability, can be stored together.

---

2. Stack Operations Using Arrays

1. **What is a stack?**
   A stack is a linear data structure that follows **LIFO – Last In, First Out**.

2. **What are the basic stack operations?**
   Push, Pop and Peek/Top.

3. **What is stack overflow?**
   Overflow occurs when we try to push an element into a full stack.

4. **What is stack underflow?**
   Underflow occurs when we try to pop an element from an empty stack.

5. **How is a palindrome checked using a stack?**
   Characters are pushed into the stack and then popped in reverse order and compared with the original string.

---

3. Printer Queue Simulation Using Linear Queue

1. **What is a queue?**
   A queue is a linear data structure that follows **FIFO – First In, First Out**.

2. **What are the main queue operations?**
   Enqueue and Dequeue.

3. **What are front and rear used for?**
   front indicates the deletion position, while rear indicates the insertion position.

4. **What is queue overflow?**
   It occurs when an insertion is attempted in a full queue.

5. **What is the disadvantage of a linear queue?**
   Empty positions created after deletion cannot be reused when rear has already reached the last array position. A circular queue solves this problem.

---

4. Singly Linked List of Student Records

1. **What is a linked list?**
   A linked list is a dynamic data structure consisting of nodes connected using pointers.
2. **What does a node in a singly linked list contain?**
   It contains data and a pointer to the next node.
3. **What is the purpose of the next pointer?**
   It stores the address of the next node.
4. **What is the advantage of a linked list over an array?**
   A linked list can grow or shrink dynamically and does not require contiguous memory.
5. **What is the time complexity of traversing a linked list?**
   O(n), because every node may need to be visited.

---

5. Binary Tree – Level-Order Creation and Traversals

1. **What is a binary tree?**
   A binary tree is a tree in which each node has at most two children: left and right.
2. **What are the three depth-first traversals?**
   Preorder, Inorder and Postorder.
3. **What is the order of preorder traversal?**
   Root → Left → Right.
4. **What is the order of inorder traversal?**
   Left → Root → Right.
5. **Which data structure is commonly used for level-order traversal?**
   A **queue**, because nodes are processed level by level.

---

6. Graph Representation and Traversal – DFS/BFS

1. **What is a graph?**
   A graph consists of vertices (nodes) and edges connecting those vertices.
2. **What is an adjacency matrix?**
   It is a two-dimensional array used to represent connections between vertices.

3. **What is DFS?**
   Depth First Search explores a vertex as deeply as possible before backtracking.

4. **What data structure is used in BFS?**
   A **queue**.

5. **What data structure is commonly used in DFS?**
   A **stack**, either explicitly or implicitly through recursion.

Part-B: Open-Ended Experiments

1. Sparse Matrix Addition Using 3-Tuple Representation

1. **What is a sparse matrix?**
   A matrix in which most of the elements are zero.

2. **Why is 3-tuple representation used?**
   It stores only non-zero elements, thereby saving memory.

3. **What information does each 3-tuple contain?**
   Row number, column number and value.

4. **What does the first row of the 3-tuple represent?**
   It generally stores **number of rows, number of columns and number of non-zero elements**.

5. **What is the main advantage of sparse representation?**
   It reduces memory usage and can make operations on sparse data more efficient.

---

2. Infix to Postfix Conversion Using Stack

1. **What is an infix expression?**
   An expression where the operator is written between operands, such as A+B.

2. **What is a postfix expression?**
   An expression where the operator comes after its operands, such as AB+.

3. **Why is a stack used for infix-to-postfix conversion?**
   A stack temporarily stores operators according to their precedence and associativity.

4. **What is the precedence order of common operators?**
   ^ has higher precedence, followed by * / %, followed by + -.

5. **Convert (A+B)*C into postfix.**
   AB+C*

---

3. Circular Queue Operations

1. **What is a circular queue?**
   A queue in which the last position is logically connected to the first position.

2. **Why is a circular queue better than a linear queue?**
   It allows previously freed positions to be reused.

3. **How is the next position calculated?**
   Using: **(index + 1) % MAX**

4. **When is a circular queue empty?**
   Typically when front == -1.

5. **When is a circular queue full?**
   When (rear + 1) % MAX == front.

---

4. Doubly Linked List of Employee Records with DEQ

1. **What is a doubly linked list?**
   A linked list in which every node contains pointers to both the previous and next nodes.

2. **What is DEQ?**
   DEQ stands for **Double-Ended Queue**, where insertion and deletion can be performed at both ends.

3. **What are the two pointers in a doubly linked-list node?**
   prev and next.

4. **What is an advantage of a doubly linked list?**
   It allows traversal in both forward and backward directions.

5. **What is the time complexity of insertion at either end of a DEQ when front and rear pointers are maintained?**
   O(1).

---

5. Binary Search Tree Operations

1. **What is a Binary Search Tree?**
   A binary tree in which keys smaller than the root are stored in the left subtree and larger keys in the right subtree.

2. **What is the important property of inorder traversal in a BST?**
   It produces the elements in **sorted order**.

3. **What is the average search complexity in a balanced BST?**
   O(log n).

4. **What is the worst-case search complexity of a BST?**
   O(n), when the tree becomes skewed.

5. **What happens when duplicate keys are inserted?**
   The implementation must define a policy; commonly duplicates are ignored or consistently placed on one side.

---

6. Employee Record Management Using Hashing – Linear Probing

1. **What is hashing?**
   Hashing is a technique that maps a key to an index in a hash table using a hash function.

2. **What is a hash function?**
   A function that converts a key into a table index. For example, key % table_size.

3. **What is a collision?**
   A collision occurs when two different keys produce the same hash index.

4. **What is linear probing?**
   It resolves collisions by checking consecutive table positions until an empty position is found.

5. **What is the average search complexity of hashing?**
   Approximately **O(1)** with a good hash function and suitable load factor; the worst case can be **O(n)**.