Data Structures Laboratory

2024-25

Prepared
by
G Vannurswamy
Assistant Professor

Data Structures and Applications

1. Dynamic Array Operations

Objective: To understand and implement dynamic arrays, performing insertion, deletion, and display operations.

Description: Dynamic arrays are arrays that can change in size during runtime. This program involves operations to manage the size of the array and manipulate its elements.

Operations:

- **Insertion**: Add a new element at a specified position. If the array is full, resize it to accommodate new elements.
- **Deletion**: Remove an element from a specified position and shift subsequent elements.
- **Display**: Show the current elements of the array and its size.
- **Exit**: Terminate the program, ensuring proper cleanup of allocated memory.

Example: Suppose we start with an array of size 5 and insert elements 10, 20, and 30. If we then add more elements, the array might resize to accommodate them, demonstrating dynamic resizing.

Analysis:

- **Dynamic Resizing**: Ensures that the array can grow as needed, typically doubling its size when full.
- **Complexity**: Insertion operations are O(1)O(1)O(1) on average, but resizing is O(n)O(n)O(n), where nnn is the number of elements.
- **Efficiency**: Proper handling of memory allocation and resizing ensures efficient use of resources.

---

2. Sparse Matrix Operations

Objective: To manage and perform operations on sparse matrices, focusing on creation and transposition.

Description: Sparse matrices have mostly zero values and are stored efficiently to save space. The program involves creating a sparse matrix and computing its transpose.

Operations:

- **Create Sparse Matrix**: Store only non-zero elements along with their positions.
- **Transpose**: Flip the matrix over its diagonal, swapping rows with columns.

Example: Consider a matrix with non-zero elements at positions (0,1), (1,2), and (2,0). The transpose will move these to (1,0), (2,1), and (0,2).

Analysis:

- **Space Efficiency**: Sparse matrices save memory by storing only essential data.
- **Complexity**: Creation and transposition operations are O(n)O(n)O(n), where nnn is the number of non-zero elements.
- **Usage**: Ideal for large matrices with few non-zero elements.

---

3. Stack Operations Using Array

Objective: To implement and manage stack operations using an array, including handling overflow and underflow.

Description: Stacks follow the Last In, First Out (LIFO) principle. Operations include pushing, popping, and checking for overflow or underflow conditions.

Operations:

- **Push**: Add an element to the top of the stack.
- **Pop**: Remove the top element from the stack.
- **Overflow/Underflow**: Handle cases where the stack is full or empty.
- **Display**: Show current stack elements.

Example: Push elements 10, 20, 30 onto the stack. Pop the top element to remove 30. Display the stack to show 10 and 20.

Analysis:

- **Stack Operations**: Push and pop are efficient operations with O(1)O(1)O(1) time complexity.
- **Overflow/Underflow**: Managed by checking stack size limits and current status.
- **Efficiency**: Array-based implementation provides constant time operations for typical stack use cases.

---

4. Infix to Postfix Conversion

Objective: To convert infix expressions to postfix expressions, managing parentheses and operator precedence.

Description: Infix expressions have operators between operands, while postfix expressions place operators after their operands. The conversion is managed using a stack.

Operations:

- **Conversion**: Apply the Shunting Yard algorithm to reorder operators and operands.
- **Parentheses Handling**: Ensure proper placement of operators and manage nested parentheses.

Example: Convert the infix expression `3 + (2 * 5)` to postfix notation `3 2 5 * +`.

Analysis:

- **Complexity**: Conversion is O(n)O(n)O(n), where nnn is the length of the infix expression.
- **Handling Parentheses**: Requires stack management to respect operator precedence and grouping.
- **Use Cases**: Useful for evaluating expressions and implementing compilers.

---

5. Evaluation of Postfix Expression

Objective: To evaluate postfix expressions using a stack for operations with single-digit operands.

Description: Postfix expressions place operators after their operands. Evaluation involves using a stack to compute results.

Operations:

- **Evaluation**: Process operands and operators using a stack to calculate the expression's result.

Example: Evaluate the postfix expression `5 3 4 * +` which equals `5 + (3 * 4) = 17`.

Analysis:

- **Complexity**: Evaluation is O(n)O(n)O(n), where nnn is the number of characters in the postfix expression.
- **Stack Usage**: Utilizes stack operations to store intermediate results and apply operators.
- **Efficiency**: Provides a straightforward method for evaluating expressions without precedence concerns.

---

6. Circular Queue Operations

Objective: To implement circular queue operations using an array, handling insertion, deletion, and overflow/underflow.

Description: A circular queue allows the end of the queue to wrap around to the beginning. Operations include adding and removing elements while managing circular indexing.

Operations:

- **Insert**: Add an element to the rear of the queue.
- **Delete**: Remove an element from the front of the queue.
- **Overflow/Underflow**: Handle cases where the queue is full or empty.
- **Display**: Show current queue elements and status.

Example: Enqueue elements 1, 2, 3 into a circular queue. Dequeue an element, then display the queue status to reflect remaining elements.

Analysis:

- **Circular Nature**: Efficiently uses array space by wrapping around.
- **Complexity**: Insert and delete operations are O(1)O(1)O(1) with proper index management.
- **Handling Overflow/Underflow**: Proper index checks prevent errors and ensure queue operations are safe.

---

7. Singly Linked List Operations

Objective: To implement operations on a singly linked list, including insertion and deletion at both ends.

Description: A singly linked list consists of nodes where each node points to the next. This lab will focus on managing nodes and performing list operations.

Operations:

- **Create SLL**: Insert nodes at the front.
- **Display**: Show the list status and count nodes.
- **Insertion/Deletion at End**: Add or remove nodes from the end.
- **Insertion/Deletion at Front**: Add or remove nodes from the front.

Example: Create a list by inserting nodes with student data at the front. Display the list to show data, and then delete a node from the end.

Analysis:

- **Linked List Operations**: Insertion and deletion operations are O(1)O(1)O(1) if pointers are properly managed.
- **Traversal**: To count nodes, a traversal of the list is required, which is O(n)O(n)O(n).
- **Flexibility**: Allows dynamic data management and efficient insertions/deletions.

---

8. Doubly Linked List Operations

Objective: To implement operations on a doubly linked list, allowing insertion and deletion at both ends.

Description: A doubly linked list has nodes with pointers to both previous and next nodes, allowing bidirectional traversal and operations.

Operations:

- **Create DLL**: Insert nodes at the end.
- **Display**: Show the list status and count nodes.
- **Insertion/Deletion at End**: Manage nodes at the end.
- **Insertion/Deletion at Front**: Manage nodes at the front.

Example: Insert nodes at the end of a doubly linked list. Display the list, then delete a node from the front.

Analysis:

- **Bidirectional Pointers**: Provides efficient operations at both ends.
- **Complexity**: Insertion and deletion are O(1)O(1)O(1) with proper pointer adjustments.
- **Use Cases**: Useful for scenarios requiring efficient bidirectional traversal.

---

9. Binary Search Tree (BST) Operations

Objective: To implement operations on a Binary Search Tree (BST), including creation, traversal, and searching.

Description: A BST maintains a sorted order of elements, allowing efficient insertion, deletion, and search operations.

Operations:

- **Create BST**: Insert nodes to form a BST with given values.
- **Traversal**: Perform Inorder, Preorder, and Postorder traversals.
- **Search**: Locate a specific value in the BST.

Example: Create a BST with values 6, 9, 5, 2, 8, 15, 24, 14, 7, 8, 5, 2. Perform Inorder traversal to get the sorted order.

Analysis:

- **BST Properties**: Ensures ordered data and efficient search operations.

- **Traversal**: Each traversal method (Inorder, Preorder, Postorder) has O(n)O(n)O(n) complexity.
- **Search**: Efficient with average time complexity of O(log n)O(\log n)O(logn), but can degrade to O(n)O(n)O(n) in the worst case.

---

10. Graph Operations

Objective: To manage graph data structures using adjacency matrices, and perform traversal operations.

Description: Graphs represent relationships between entities. Adjacency matrices provide a way to store graph connections.

Operations:

- **Create Graph**: Use an adjacency matrix to represent edges between nodes.
- **Traversal**: Implement Breadth-First Search (BFS) and Depth-First Search (DFS).

Example: Create a graph with nodes and edges. Perform BFS and DFS to explore nodes and edges.

Analysis:

- **Adjacency Matrix**: Space complexity is O(V2)O(V^2)O(V2), where VVV is the number of vertices.
- **Traversal**: BFS and DFS have O(V+E)O(V + E)O(V+E) complexity, where EEE is the number of edges.
- **Use Cases**: Suitable for network analysis, path finding, and scheduling problems.

---

11. Hashing and Collision Resolution

Objective: To understand and implement hashing techniques with collision resolution using linear probing in a hash table.

Description: Hashing is a technique used to efficiently store and retrieve data using a hash function that maps keys to specific locations in a hash table. However, collisions occur when multiple keys hash to the same index. This program focuses on creating a hash table, handling collisions using linear probing, and analyzing the efficiency of these operations.

Operations:

1. **Create Hash Table**:

- **Hash Function**: Use a simple hash function H(K)=Kmod mH(K) = K \mod mH(K)=Kmodm, where KKK is the key and mmm is the size of the hash table. This function maps the key to an index in the table.
- **Initialization**: The hash table is initialized with empty slots. Keys are inserted into the table based on their computed hash values.

2. **Handle Collisions**:
   - **Linear Probing**: When a collision occurs (i.e., when the desired slot is already occupied), the program finds the next available slot by incrementing the index by 1 (i.e., moving to the next position) until an empty slot is found.
   - **Efficiency Consideration**: Linear probing continues until an empty slot is found or the table is deemed full. If the hash table becomes too full, rehashing or resizing may be necessary.
3. **Search and Retrieval**:
   - **Search**: Given a key, the program computes its hash and checks the corresponding index. If the key is not found at that index due to collision resolution, linear probing is used to search adjacent slots.
   - **Retrieval**: Upon locating the key, its associated data is retrieved and displayed.
4. **Example**:
   - Consider a hash table with size m=10m = 10m=10 and the following keys: 23, 43, 13, 27, and 37.
   - The hash function H(K)=K mod 10 H(K) = K mod 10 H(K)=K mod 10 would map these keys to indices 3, 3, 3, 7, and 7 respectively.
   - Collisions occur for keys 43, 13, and 37, as they map to already occupied slots (3 and 7).
   - Linear probing resolves these collisions by placing 43 at index 4, 13 at index 5, and 37 at index 8.

Analysis:

- **Efficiency**:
  - **Time Complexity**:
    - **Insertion and Search**: In the best case, both operations are O(1)O(1)O(1) when there are no collisions. In the worst case, where collisions lead to multiple probes, the complexity can approach O(n)O(n)O(n) for a nearly full table.
  - **Space Complexity**: The hash table requires O(m)O(m)O(m) space, where mmm is the number of slots in the table.
- **Collision Resolution**:
  - **Linear Probing**: While simple and easy to implement, linear probing can lead to clustering, where a group of adjacent slots are filled, increasing the number of probes needed for subsequent insertions and searches.
  - **Primary Clustering**: This is a potential drawback of linear probing, where clusters of filled slots grow, degrading performance.

```
/* 1. Write a menu driven C Program to create a dynamic array of n elements and
Perform the following operations
     a) insert a new element at a specified Position
     b) delete an element at a specified position
     c) Display
     d) Exit
*/
```

Course Objectives

1. **Understanding Dynamic Memory Allocation**:
   - Students will learn how to create and manage dynamic arrays using pointers in C, emphasizing the differences between static and dynamic memory allocation.
2. **Developing Problem-Solving Skills**:
   - Students will practice breaking down problems into smaller tasks and implementing algorithms to perform operations such as insertion, deletion, and traversal on arrays.
3. **Enhancing Programming Logic**:
   - Students will strengthen their logic-building skills by developing menu-driven programs that handle user input and perform specified operations on data structures.
4. **Mastering Array Manipulation**:
   - Students will gain proficiency in manipulating array elements, understanding array indexing, and managing array boundaries while performing insertions and deletions.

```c
#include <stdio.h>
int main()
{
    int *p, n, ele, ch, i, pos; // p is pointer to create a dynamic array
    printf("Enter number of elements to create an Array:\t");
    scanf("%d", &n);
    p = malloc(n * sizeof(int)); // use malloc to create a Dynamic array
    printf("Dynamic Array Created.\n");
    printf("Enter %d elements\n ", n);
    for (i = 0; i < n; i++) // Read n the elements to array
    {
        scanf("%d", &p[i]);
    }
```

```
    while (1)
    {
        printf("\n 1.Insert\n 2.delete\n 3.display \n 4.Exit\n Enter your
choice:\t");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
            printf("\n Enter element & Pos(0 to %d) to insert:\t", n - 1);
            scanf("%d%d", &ele, &pos);
            realloc(p, (n+1) * sizeof(int));   // increase the size of array by 1
            n = n + 1;                         // update new size
            for (i = n - 1; i >= pos; i--) // Start moving all the elements to the
next positions
            {
                p[i] = p[i - 1];
            }
            p[pos] = ele; // insert the new element at specied position
            break;
        case 2:
            printf("Enter Position(0 to %d) to delete:\t", n - 1);
            scanf("%d", &pos);
            for (i = pos + 1; i < n; i++) // delete the position element by moving
next element of pos to prevvious pos
            {
                p[i - 1] = p[i];
            }
            n = n - 1; // Update the count total elements
            break;
        case 3:
            printf("\n Array Elements Are:\n");
            for (i = 0; i < n; i++)
```

```
            {
                printf("%d\t", p[i]);
            }
            break;
        case 4:
            exit(0);
        }
    }
    return 0;
}
```

OUTPUT:

Course Outcomes

1. **Demonstrate Proficiency in Dynamic Memory Management**:
   - Students will be able to allocate, reallocate, and deallocate memory dynamically using pointers and functions such as `malloc()`, `realloc()`, and `free()`.
2. **Implement Array Operations Efficiently**:
   - Students will be able to implement insertion and deletion of elements at specified positions in an array, ensuring proper memory handling and avoiding overflow or underflow conditions.
3. **Create and Navigate Menu-Driven Programs**:
   - Students will be able to design and implement menu-driven programs that allow users to perform various operations on dynamic arrays through a simple interface.

```
/*
   2. Write a menu driven program for the following operations
   a. Create a sparse Matrix
   b. Transpose of sparse Matrix
   c. Exit
*/
```

Course Objectives

1. **Understanding Sparse Matrices**:
   - Students will learn the concept of sparse matrices, where most elements are zero, and the importance of efficient storage and manipulation of such matrices in memory.
2. **Developing Skills in Matrix Representation**:
   - Students will gain proficiency in representing sparse matrices using data structures like arrays or linked lists, optimizing storage and processing.
3. **Enhancing Problem-Solving Abilities**:
   - Students will practice implementing algorithms to create, traverse, and transpose sparse matrices, improving their ability to solve complex problems using efficient methods.
4. **Mastering Matrix Manipulation Techniques**:
   - Students will learn and implement techniques to transpose sparse matrices, understanding the mathematical and logical principles behind matrix operations.

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

#define MAX 100

struct term {
    int row;
    int col;
    int value;
};
struct term sparse[MAX],trans[MAX];
int size;   // Total num of values..
```

```
void create();
void transpose();
void display(int values,struct term matrix[]);
int main() {
   int choice;
    while(1) {
        printf("\nMenu:\n");
        printf("1. Create Sparse Matrix\n");
        printf("2. Transpose of Sparse Matrix\n");
       // printf(". Display Sparse Matrix\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                create();
                break;
            case 2:
                transpose();
                break;
            case 3:
                exit(0);
           }
    }
    return 0;
}
```

```
void create() {
    int matrix[10][10];
    int i,rows,cols,values; // Starting index from 1 since 0 is used for
matrix dimensions
    printf("\nEnter number of Rows,Columns and number of Values :");
    scanf("%d%d%d",&rows,&cols,&values);
    sparse[0].row = rows;
    sparse[0].col = cols;
    sparse[0].value = values; // Initially no non-zero elements
   for(i=1;i<=values;i++)
   {
    printf("\n Enter row,col and value:");
    scanf("%d%d%d",&sparse[i].row,&sparse[i].col,&sparse[i].value);
   }
   display(values,sparse);
}

void transpose() {
    int i,values;
    // trans[0].row = sparse[0].col;
    // trans[0].col = sparse[0].row;
    // trans[0].value = sparse[0].value;
    values=sparse[0].value;
    for(i=0;i<=sparse[0].value;i++)
    {
        trans[i].row=sparse[i].col;
        trans[i].col=sparse[i].row;
        trans[i].value=sparse[i].value;
```

```
    }
        display(values,trans);
}
void display(int values,struct term a[]) {
    int i;
    printf("\n\t Row\tColumn\tValue\n");
    for (i = 0; i <= values; i++) {
        printf("a[%d]: %d\t%d\t%d\n",i, a[i].row, a[i].col, a[i].value);
    }
}
```

Course Outcomes

1. **Demonstrate Understanding of Sparse Matrix Concepts**:
   - Students will be able to explain the concept of sparse matrices, including the benefits of using sparse matrix representations in terms of memory and computational efficiency.
2. **Efficiently Represent Sparse Matrices**:
   - Students will be able to implement and store sparse matrices using arrays or linked lists, ensuring that only non-zero elements are stored, reducing memory usage.
3. **Implement Matrix Operations**:
   - Students will be able to perform operations such as creating and transposing sparse matrices, understanding the underlying logic and algorithms involved.

```
/*
3.Develop a menu driven Program in C for the following operations on STACK of
Integers (Array Implementation of Stack with maximum size MAX)
a. Push an Element on to Stack
b. Pop an Element from Stack
c. Demonstrate Overflow and Underflow situations on Stack
d. Display the status of Stack
e. Exit
 Support the program with appropriate functions for each of the above operations
*/
```

Course Objectives

1. **Understanding Stack Data Structure**:
   - Students will learn the fundamental concepts of the stack data structure, including its LIFO (Last In, First Out) principle, and the various operations that can be performed on stacks.
2. **Mastering Array Implementation of Stacks**:
   - Students will gain proficiency in implementing stacks using arrays, learning how to manage a fixed-size stack efficiently and understanding the limitations of this approach.
3. **Developing Problem-Solving Skills**:
   - Students will practice implementing stack operations such as push, pop, and checking for overflow/underflow conditions, enhancing their problem-solving and algorithmic thinking.

```c
#include <stdio.h>
#define MAX 6
int stack[MAX], ele, num, top = -1;
void push(int);
int pop();
void stakstatus();
void display();
int main()
{
    int ch;
    while (1)
    {
```

```
        printf("\n1.Push \n2.Pop \n3.Stack Status \n4.Display\n 5.Exit \n Enter
Your choice: ");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
            printf("\n Enter element to Push: ");
            scanf("%d", &ele);
            push(ele);
            break;
        case 2:
            ele = pop();
            printf("\n Popped element from stack: %d", ele);
            break;
        case 3:
            stakstatus();
            break;
        case 4:
            display();
            break;
        case 5:
            exit(0);
        }
    }
}
void push(int ele)
{
    if (top == MAX - 1) //  if top==MAX-1 stack is full ...
    {
        printf("\n Stack is Overflow...\n");
    }
    else
```

```
    {
        stack[++top] = ele; // Increment top and push element to stack
    }
}
int pop()
{
    if (top == -1) // if top=-1 stack is empty you cannot pop element
    {
        printf("\n Stack is underflow! \n");
    }
    else
    {
        return stack[top--]; // pop last element inserted from stack
    }
}

void stakstatus()
{
    if (top == MAX - 1) // Check the condition to stack full or not
    {
        printf("Stack is Full!");
    }
    display();
}

void display()
{
    int i;
    if (top == -1)
    {
        printf("Stack is empty!\n");
    }
```

```
    else
    {
        printf("Stack eles are \n");
        for (i = top; i >= 0; i--)
        {
            printf("%d \n", stack[i]);
        }
    }
}
```

O/P:

Course Outcomes

1. **Demonstrate Understanding of Stack Concepts**:
   - Students will be able to explain the stack data structure, including its properties, operations, and typical use cases in computing.
2. **Implement Stack Operations Using Arrays**:
   - Students will be able to implement a stack using an array, efficiently performing operations such as push and pop while managing the fixed size of the array.
3. **Effectively Handle Stack Overflow and Underflow**:
   - Students will be able to detect and handle overflow (when attempting to push an element onto a full stack) and underflow (when attempting to pop an element from an empty stack) conditions, ensuring the program remains robust.

4. Develop a Program in C for converting an Infix Expression to Postfix Expression. Program should support for both parenthesized and free parenthesized expressions with the operators: +, -, *, /, % (Remainder), ^ (Power) and alphanumeric operands.

Course Objectives

1. **Understanding Expression Notations**:
   - Students will learn the different types of expression notations—Infix, Postfix (Reverse Polish Notation), and Prefix—focusing on their applications and conversions between them.
2. **Mastering Stack-Based Algorithms**:
   - Students will gain proficiency in using stack data structures to implement algorithms for converting infix expressions to postfix expressions, leveraging the stack's properties for handling operators and parentheses.
3. **Handling Operator Precedence and Associativity**:
   - Students will develop an understanding of operator precedence and associativity rules, and learn how to apply these rules correctly during the conversion process.
4. **Implementing Parenthesized and Non-Parenthesized Expressions**:
   - Students will learn to handle both parenthesized and free parenthesized expressions, ensuring the program can correctly process and convert expressions with various types of operators and operands.

```c
#include<stdio.h>
#include<ctype.h>
char stack[20];
int top=-1;
void push(char ele);
char pop();
int priority(char sym);
int main()
{
    int i=0;
    char exp[20];
    char sym,ele;
    printf("Enter valid Infix expression:");
    scanf("%s",exp);
    printf("\n Postfix:");
    for(i=0;exp[i]!='\0';i++)   // read each char from infix
    {
        sym=exp[i];
        if(isalnum(sym))    // check is alpha numeric
```

```
            printf("%c ",sym);   // print output if it is operand alphabet
        else if(sym=='(')
            push(sym);      // push symbol if it lparenthsis
        else if(sym==')')   // if rparenthesis encountered
        {
          while((ele=pop())!='(') // Pop all the elements from stack till
                printf("%c ",ele); // left parenthesis encountered & print on
output
        }
        else
        {
            // check priority of incoming symbol


            while(priority(stack[top])>=priority(sym))
                printf("%c ",pop()); // and stack[top] symbol print on output
            push(sym);     // Push the symbol
        }
    }
        while(top!=-1)   // Pop remaining all the ele from stack and print
        {                 // on output..
            printf("%c ",pop());
        }

    return 0;
}
void push(char ele)
{
    stack[++top]=ele;
}
char pop()
{
    return stack[top--];
}
int priority(char sym)  // initialize priorities
{
```

```
    if(sym=='(')
        return 0;
    if(sym=='+'|| sym=='-')
        return 1;
    if(sym=='*'|| sym=='/'|| sym=='%')
        return 2;
    if(sym=='^')
        return 3;
    return 0;
}
```

O/P:

Course Outcomes

1. **Demonstrate Understanding of Expression Notations**:
   - Students will be able to explain the differences between infix, postfix, and prefix notations, and describe the scenarios where each notation is used.
2. **Implement Infix to Postfix Conversion Using Stacks**:
   - Students will be able to write a C program that efficiently converts infix expressions to postfix expressions using stack data structures, correctly handling operators, operands, and parentheses.
3. **Apply Operator Precedence and Associativity Rules**:
   - Students will be able to implement the correct application of operator precedence and associativity rules during the conversion process, ensuring the output postfix expression is accurate.
4. **Handle Parenthesized Expressions**:
   - Students will be able to process both parenthesized and free parenthesized infix expressions, ensuring the correct handling of parentheses in the conversion to postfix notation.

```
/*
5.Develop a Program in C for the following Stack Applications
a. Evaluation of Suffix expression with single digit operands and operators:
   +, -, *, /, %,^
*/
```

Course Objectives

1. **Understanding Postfix (Suffix) Notation**:
   - Students will learn the concept of postfix (suffix) notation, including how it differs from infix notation, and why it is useful in certain computational contexts such as expression evaluation.
2. **Mastering Stack-Based Expression Evaluation**:
   - Students will gain proficiency in using the stack data structure to evaluate postfix expressions, leveraging the LIFO (Last In, First Out) property of stacks to process operands and operators.
3. **Implementing Arithmetic Operations in Postfix Evaluation**:
   - Students will learn how to perform basic arithmetic operations (addition, subtraction, multiplication, division, modulus, and exponentiation) during the evaluation of postfix expressions with single-digit operands.

```
#include<stdio.h>
#define MAX 10
int stack[MAX],top=-1;
void push(int);
int pop();
void eval(int op1,char sym,int op2);
int main()
{
    int i=0,op1,op2;
    char exp[20];
    char sym;
    printf("Enter postfix expression:\t");
    scanf("%s",exp);
    for(i=0;exp[i]!='\0';i++)
    {
        sym=exp[i];
        if(isdigit(sym))    // Check given symbol is a digit or operator
        {
            push(sym-'0'); // Symbol converted to int
        }
```

```
        else{
            op2=pop();      // Pop the top ele as op2
            op1=pop();      // Pop the next top ele as op1
            eval(op1,sym,op2); // Call eval function to find res..
        }
    }
    printf("Result of given expression=%d",pop()); // Last ele in stack is res
    return 0;
}


void push(int ele)
{
    stack[++top]=ele;
}
int pop()
{
    return stack[top--];
}
void eval(int op1,char sym,int op2)
{
    int res;
    switch(sym)
    {
        case '+':res= op1+op2; // Calculate addition of Op1+Op2
                 push(res);  // Result Push it into stack
                 break;
        case '-':res= op1-op2;
                 push(res);
                 break;
        case '*':res= op1*op2;
                 push(res);
                 break;
        case '/':res= op1/op2;
                 push(res);
                 break;
```

```
        case '%':res= op1%op2;
                 push(res);
                 break;
    }
}
```

O/P:

Course Outcomes

1. **Demonstrate Understanding of Postfix Notation**:
   - Students will be able to explain the postfix (suffix) notation and describe its advantages in the context of expression evaluation, particularly in eliminating the need for parentheses.
2. **Implement Postfix Expression Evaluation Using Stacks**:
   - Students will be able to write a C program that evaluates postfix expressions using a stack, processing single-digit operands and applying arithmetic operators correctly.
3. **Perform Arithmetic Operations in Postfix Evaluation**:
   - Students will be able to apply arithmetic operations (including addition, subtraction, multiplication, division, modulus, and exponentiation) to operands within a postfix expression during evaluation.

```
/*
6. Design, Develop and Implement a menu driven Program in C for the following
operations on Circular QUEUE of Characters (Array Implementation of Queue with
maximum size MAX)
a. Insert an Element on to Circular QUEUE
b. Delete an Element from Circular QUEUE
c. Demonstrate Overflow and Underflow situations on Circular QUEUE
d. Display the status of Circular QUEUE
e. Exit
Support the program with appropriate functions for each of the above operations.*/
```

Course Objectives

1. **Understanding Circular Queue Data Structure**:
   - Students will learn the concept of circular queues, including how they differ from linear queues, and the benefits of using circular queues for efficient memory utilization.
2. **Mastering Array Implementation of Circular Queues**:
   - Students will gain proficiency in implementing circular queues using arrays, learning to handle the circular nature of the queue through appropriate indexing and pointer manipulation.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX 5
char cqueue[MAX], element;
int front = 0, rear = -1, count = 0; // count set to 0 Initially Q Empty.
void insert(char ele);
void delete();
void display();
int main()
{
    int ch;
    while(1)
    {
        printf("\n1.insert\n2.delete \n3.display \n4.exit\n Enter Your Choice:");
        scanf("%d", &ch);
        switch (ch)
```

```
        {
        case 1:printf("\n Enter a char element to insert:");
               scanf(" %c",&element);  // Space is mandatory Before %c
               insert(element);   // Call insert function
            break;
        case 2:
            delete ();
            break;
        case 3:
            display();
            break;
        case 4:
            return;
        }
    }
    return 0;
}
void insert(char ele)
{
    if (count == MAX)  // if count is MAX then queue is full
    {
        printf("Circular Queue is full\n");
        return;
    }
    rear = (rear + 1) % MAX; //Find rear and insert element
    cqueue[rear] = ele; // Inserting element into queue
    count++;    // Increment the count of elements
    printf("Element inserted into C-Queue. \n");
   // display();
}
void delete()
{
    if (count == 0)
    {
        printf("Circular Queue is empty\n");
        return;
```

```
    }
    printf("Element deleted from C-Queue: %c\n", cqueue[front]);
    front = (front + 1) % MAX; // Change front position after deletion
    count-=1;  // Decrement count after deleting an element.
}
void display()
{
    int i;
    if (count == 0)
    {
        printf("\n Circular Queue is empty!\n");
        return;
    }
    else
    {
        // Display elements from front and till rear
       for (i=front;i!=rear;i=(i+1)%MAX)
        {
           printf("%c \n", cqueue[i]);
        }
        printf("%c",cqueue[i]); // print last element
    }
}
```

O/P:

Course Outcomes

1. **Demonstrate Understanding of Circular Queues**:
   - Students will be able to explain the concept of circular queues, including their advantages over linear queues, particularly in terms of efficient memory usage.
2. **Implement Circular Queue Operations Using Arrays**:
   - Students will be able to write a C program that efficiently implements circular queue operations (insert, delete) using an array, handling the circular indexing correctly.
3. **Effectively Handle Overflow and Underflow Situations**:
   - Students will be able to detect and manage overflow and underflow conditions in a circular queue, ensuring the program can handle these situations without crashing or producing incorrect results.

```
/*
7. Design, Develop and Implement a menu driven Program in C for the following operations on
Singly Linked List (SLL)
of Student Data with the fields : USN, Name, Branch, rollno, PhNo
a.Create a SLL of N Students Data by using front insertion.
b.Display the status of SLL and count the number of nodes in it
c.Perform Insertion / Deletion at End of SLL
d.Perform Insertion / Deletion at Front of SLL(Demonstration of stack) e.Exit
*/
```

Course Objectives

1. **Understanding Linked List Data Structure**:
   - Students will learn the fundamental concepts of linked lists, focusing on singly linked lists (SLL), and understand how they differ from arrays in terms of memory allocation, insertion, and deletion operations.
2. **Developing Skills in Dynamic Memory Management**:
   - Students will gain proficiency in using pointers for dynamic memory allocation, learning to manage memory effectively while creating and manipulating linked list nodes.
3. **Mastering Linked List Operations**:
   - Students will learn how to perform various operations on singly linked lists, including insertion and deletion of nodes at the front and end of the list, as well as counting and displaying the nodes.
4. **Implementing Data Structures for Real-World Scenarios**:
   - Students will apply linked list concepts to store and manage real-world data, such as student information, demonstrating how data structures can be used to model and solve practical problems.

```c
#include <stdio.h>
struct student
{
    int rollno;
    char usn[20];
    char name[50];

    struct student *link;
};
typedef struct student *NODE;
int count = 0; // To count total no of nodes in a list..
NODE createNode();
void CreateSLL(); // Creating list by inserting fornt
void DisplaySLL();
void InsertFront();
void InsertEnd();
void DeleteFront();
void DeleteEnd();
NODE first = NULL;
int main()
{
    int ch;
```

```
    while(1)
    {
        printf("1.CreateSLL \n2.DisplaySLL \n3.Insert front \n4.Insert End \n5.Delete
Front\n6.Delete End \n7.Exit\nEnter Your Choice:");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
            CreateSLL();
            break;
        case 2:
            DisplaySLL();
            break;
        case 3:
            InsertFront();
            break;
        case 4:
            InsertEnd();
            break;
        case 5:
            DeleteFront();
            break;
        case 6:
            DeleteEnd();
            break;
        case 7:
            exit(0);
        }
    }
    return 0;
}
NODE createNode()
{
    // Creating student node dtails
    NODE temp;
    // Allocate memeory for temp node
    temp = malloc(sizeof(struct student));
    printf("\n Enter Roll Number:");
    scanf("%d", &temp->rollno);
    // read student details
    printf("\n Enter Usn:");
    scanf("%s", temp->usn);
    printf("\n Enter Student Name:");
    scanf("%s", temp->name);
    temp->link = NULL; // Make temp link null default
    count++;           // Increment the count new node is created
    return temp;       // return newly created node
}
void CreateSLL()
{
```

```
    int i, n;
    NODE temp;
    // Enter number students details to create SLL
    printf("\n Enter number of students:");
    scanf("%d", &n);
    // Create n number Student details SLL by inserting front
    for (i = 0; i < n; i++)
    {
        printf("Enter Student %d details:", i + 1);
        temp = createNode(); // Copy student node to temp
        if (first == NULL)   // If first is NULL no nodes in the list
        {
            first = temp; // Make first node as Temp
        }
        else
        {
            // insert front of the list store first node address to temp
            temp->link = first;
            first = temp; // Make temp as first node
        }
        printf("Student node inserted at Front of List \n");
    }
    DisplaySLL();
}
void DisplaySLL()
{
    NODE cur = first;
    if (first == NULL)
    {
        printf("\n Student List is empty!");
        return;
    }
    printf("\nStudents List:\n");
    printf("-----------------------------------------------\n");
    while (cur != NULL)
    {
        // print student information....
        printf("%d\t%s\t%s\n",cur->rollno, cur->usn, cur->name);
        cur = cur->link;
    }
    printf("\n");
    printf("Total Number of students: %d \n", count);
}
void InsertFront()
{
    NODE temp = createNode();
    temp->link = first;
    first = temp;
    DisplaySLL(); // Call CreateSLL to insert front of the list
}
```

```c
void InsertEnd()
{
    NODE temp, cur;           // To create node temp node taken as reference
    cur = first;              // make current as first
    while (cur->link != NULL) // Go to end of the list
    {
        cur = cur->link; // Move current node to next node
    }
    temp = createNode(); // Create Node and copy to temp
    cur->link = temp;    // Link new node to last node
    printf("Studnet Node inserted at end of the list!\n");
    DisplaySLL(); // Display List
}
void DeleteFront()
{
    NODE cur;
    cur = first->link; // Make second node as current
    free(first);       // delete first node
    first = cur;       // make currrent node as first..
    /* or */
    // first=first->link;  // Make second as first node...
    printf("Node deleted front of the list!\n");
    count--; // decrement the node when node is deleted
    DisplaySLL();
}
void DeleteEnd()
{
    NODE cur, prev;    // Copy second last node in prev
    cur = first;       // Make current as first for reference
    if (first == NULL) // If first is null list is empty
    {
        printf("Student list is empty!\n");
        return;
    }
    if (first->link == NULL) // if list contains one node delete it
    {
        free(first); // delete first node
    }
    while (cur->link != NULL) // Move to end of the list
    {
        prev = cur;      // Store second last node in prev node
        cur = cur->link; // Cur is last node..
    }
    free(cur);         // delete the last node..
    prev->link = NULL; // make prev link is null now prev is last node
    printf("Node deleted end of the list!\n");
    count--; // decrement the count any node is deleted...
    DisplaySLL();
}
```

O/P: Course Outcomes

1. **Demonstrate Understanding of Singly Linked Lists**:
   - Students will be able to explain the structure and functioning of singly linked lists, including how nodes are dynamically allocated and linked using pointers.
2. **Implement Linked List Operations Using Pointers**:
   - Students will be able to write a C program that efficiently implements the creation, insertion, deletion, and traversal of nodes in a singly linked list using dynamic memory allocation.
3. **Perform Front and End Operations on SLL**:
   - Students will be able to perform insertion and deletion of nodes at both the front and the end of a singly linked list, demonstrating their understanding of pointer manipulation and list traversal.
4. **Manage and Display Real-World Data Using SLL**:
   - Students will be able to model and manage student data using a singly linked list, effectively displaying the list's contents and counting the number of nodes to provide meaningful output.

```
/*
8.Develop a menu driven program in C for the following operations on Doubly Linked List
(DLL) of Employee Data with the fields: ssn, Name, Salary
a. Create a DLL of N Employees Data by using end insertion.
b. Display the status of DLL and count the number of nodes in it.
c. Perform Insertion and Deletion at End of DLL.
d. Perform Insertion and Deletion at Front of DLL.
e. Exit
*/
```

Course Objectives

1. **Understanding Doubly Linked List (DLL) Data Structure**:
   - Students will learn the fundamental concepts of doubly linked lists, focusing on the bidirectional nature of DLLs, where each node points to both its previous and next node.
2. **Mastering Linked List Operations with Pointers**:
   - Students will gain proficiency in using pointers for dynamic memory management, learning to implement operations such as insertion and deletion at both the front and end of a doubly linked list.
3. **Implementing Real-World Data Structures**:
   - Students will learn to model and manage real-world data, such as employee records, using doubly linked lists, demonstrating the application of data structures in solving practical problems.

```c
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
// C Structure for employee...
struct Employee
{
    struct Employee *llink; // Left link of node
    int ssn;                // Info.. of employee
    char name[50];
    float sal;
    struct Employee *rlink; // Right link of node
};
typedef struct Employee *NODE;
int count = 0;      // to count total number of employees
NODE first = NULL;  // Start with empty list..
NODE createNode();  // Creating temp node with data
void createDll();   // Creating emp DLL by inserting at end
void insertFront(); // Inserting at front of the list
void insertEnd();   // inserting at the end of the list
void deleteFront(); // Deleting node at front of list
void deleteEnd();   // delelting node at end of list
void displayDll();  // displaying the list of employees
int main()
{
    int ch;
```

```
    while (1)
    {
        printf("\n1.Create Emp DLL \n2.insert Front \n3.Insert End \n4.Delete Front \n5.Delete
End \n6.Display DLL \n7.Exit\n Enter Your Choice: ");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
            createDll(); // Create a DLL by inserting end
            break;
        case 2:
            insertFront(); // inserting front of list
            break;
        case 3:
            insertEnd(); // Inserting end of the list
            break;
        case 4:
            deleteFront(); // delete front of the list
            break;
        case 5:
            deleteEnd(); // delete end of the list
            break;
        case 6:
            displayDll();
            break;
        case 7:
            exit(0);
        }
    }
    return 0;
}
NODE createNode()
{
    NODE temp;
    temp = malloc(sizeof(struct Employee)); // allocate memory for new node
    printf("Enter emp SSN: \t");            // read all the details of emp
    scanf("%d", &temp->ssn);
    printf("Enter emp Name: \t");
    scanf("%s", temp->name);
    printf("Enter emp Salary: \t");
    scanf("%f", &temp->sal);
    temp->llink = NULL; // make new node left and right link NULL
    temp->rlink = NULL;
    count++;     // increment the count when new node is created..
    return temp; // return newly created node with data
}
void createDll()
{
    int i, n;
    NODE temp, cur; // creating DLL by inserting end of the list
```

```
    // Number of employees to create a list
    printf("\n Enter number of Employees:");
    scanf("%d", &n);
    // Create n number of employees list..
    for (i = 0; i < n; i++)
    {
        printf("Enter Employee[%d] Details:\n", i + 1);
        temp = createNode(); // Get new node with data
        if (first == NULL)   // if no nodes in list temp node is first node
        {
            first = temp;
        }
        else
        {
            cur = first;               // take cur is reference node to insert at last
            while (cur->rlink != NULL) // repeat till end of list
            {
                cur = cur->rlink; // traverse to next node
            }
            cur->rlink = temp; // attach temp to last node with rLink
            temp->llink = cur; // store second last node address in last node
        }
    }
    // call diplay of the list
    displayDll();
}
void insertFront()
{
    NODE temp;
    temp = createNode(); // Get new node with data
    first->llink = temp; // store address of temp node in first node left link
    temp->rlink = first; // insert front of the list by linking right of temp with first
    first = temp;        // Make temp as first node
    displayDll();
}
void insertEnd()
{
    NODE cur = first;
    // Create a new node with data
    NODE temp = createNode();
    // Insert at end of the function
    while (cur->rlink != NULL)
    {
        cur = cur->rlink; // Move to Last node
    }
    cur->rlink = temp; // Insert last rlink
    temp->llink = cur; // store prev node llink in last node
    count++;           // Increment the count when new node inserted..
    displayDll();
}
```

```
void deleteFront()
{
    NODE cur;
    cur = first->rlink; // store second node in current node
    free(first);        // delete front i.e first node
    first = cur;        // make second node as first node after deletion
    //   (or) use the below statements
    // first->llink=NULL;
    // first=first->rlink;
    first->llink = NULL; //  Make first node left link NULL
    count--;             // when node is deleted
    displayDll();
}
void deleteEnd()
{
    NODE cur, prev;            // Prev stores the second last node
    cur = first;               // take cur as reference to delete last node
    while (cur->rlink != NULL) // repeat till end of the list
    {
        prev = cur;       // Store second last node in cur node
        cur = cur->rlink; // Move to next node
    }
    free(cur);          // delete the last Node
    prev->rlink = NULL; // Make last node rlink null to get end of the list.
    count--;            // decrement the count when node is deleted
    displayDll();
}
void displayDll()
{
    int count = 0;
    NODE cur;          // take cur node as reference to populate data
    cur = first;       // Make use of cur as first
    if (first == NULL) // if first is null list is empty...
    {
        printf("\n List is Empty!");
        return;
    }
    // Display the heading part of details....
    printf("\n SSN \t Name \t\t Salary  \n");
    while (cur != NULL)
    {
        printf("%d \t %s \t\t %f  \n ", cur->ssn, cur->name, cur->sal);
        cur = cur->rlink; // move to the next node till end of list..
        count++;          // increment the count to get total count of nodes
    }
    printf("\n Total Num of employees:%d\n ", count); // Dispaly total count of Emps...
}
```

O/P:

Course Outcomes

1. **Demonstrate Understanding of Doubly Linked Lists**:
   - Students will be able to explain the structure and functioning of doubly linked lists, including how nodes are dynamically allocated and linked using two pointers (previous and next).
2. **Implement DLL Operations Using Pointers**:
   - Students will be able to write a C program that efficiently implements the creation, insertion, deletion, and traversal of nodes in a doubly linked list using dynamic memory allocation.
3. **Perform Insertion and Deletion at Both Ends of DLL**:
   - Students will be able to perform insertion and deletion operations at both the front and end of a doubly linked list, demonstrating their understanding of bidirectional pointer manipulation.
4. **Manage and Display Real-World Data Using DLL**:
   - Students will be able to model and manage employee data using a doubly linked list, effectively displaying the list's contents and counting the number of nodes to provide meaningful output.

```
/*
9.Develop a menu driven Program in C for the following operations on
Binary Search Tree (BST) of Integers .
a. Create a BST of N Integers: 6, 9, 5, 2, 8, 15, 24, 14, 7, 8, 5, 2
b. Traverse the BST in Inorder, Preorder and Post Order
c. Search the BST for a given element (KEY) and report the appropriate message
d. Exit
*/
```

Course Objectives

1. **Understanding Binary Search Tree (BST) Concepts**:
   - Students will learn the fundamental concepts of Binary Search Trees (BSTs), including the properties that make BSTs efficient for search, insertion, and deletion operations.
2. **Mastering BST Operations**:
   - Students will gain proficiency in implementing key operations on BSTs, such as insertion, searching for elements, and traversing the tree using various traversal methods (Inorder, Preorder, Postorder).

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

// Define the structure of a node in the binary search tree
struct node
{
    struct node *leftchild;
    int data;
    struct node *rightchild;
};
typedef struct node *treePointer;
treePointer root = NULL; // Initialize an empty tree
treePointer createNode(int value);
treePointer insertBST(treePointer root, int value);
void inorder(treePointer root);
void preorder(treePointer root);
void postorder(treePointer root);
void search(treePointer root,int key);
int main()
```

```
{
    // Array of values to create the binary search tree
    // Can take any array values Sample values given..
    int values[] = {6, 9, 5, 2, 8, 15, 24, 14, 7, 10};
    int i, ch, key, n = 10; // n is Size of the above array...
    while (1)
    {
printf("1.Create BST\n 2.Traversals \n3.Search \n4.Exit \nEnter Your Choice:");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
         /* Create BST by inserting given elements into the tree */
            for (i = 0; i < n; i++)
            {
                root = insertBST(root, values[i]);
            }
            printf("Binary Search Tree Constructed.\n ");
            break;
        case 2:
            printf("\n Inorder: ");
            inorder(root);
            printf("\n Pre order: ");
            preorder(root);
            printf("\n Post order: ");
            postorder(root);
            printf("\n");
            break;
        case 3:
            printf("\n Enter the key to Search: ");
            scanf("%d", &key);
            search(root, key);
        case 4:
            exit(0);
        }
    }
```

```
    return 0;
}
// Function to create a new node
treePointer createNode(int value)
{
    treePointer temp = malloc(sizeof(struct node));
    temp->data = value;
    temp->leftchild = NULL;
    temp->rightchild = NULL;
    return temp;
}

// Function to insert a value into the binary search tree
treePointer insertBST(treePointer root, int value)
{
    // If the tree is empty, create a new node and make it the root
    if (root == NULL)
    {
        root = createNode(value);
    }
    // If the value is less than the root,
    //insert into the leftchild subtree
    else if (value < root->data)
    {
        root->leftchild = insertBST(root->leftchild, value);
    }
    // If the value is greater than or equal to the root,
    // insert into the rightchild subtree
    else
    {
        root->rightchild = insertBST(root->rightchild, value);
    }
    return root;
}

/* Function to perform inorder(L V R) traversal of the binary */
```

```
void inorder(treePointer root)
{
    if (root != NULL)
    {
        inorder(root->leftchild);  // Visit Left Child
        printf("%d ", root->data); // Visit Root
        inorder(root->rightchild); // Visit Right Child
    }
}
/*Function to perform preorder (V L R) traversal of the binary*/
void preorder(treePointer root)
{
    if (root != NULL)
    {
        printf("%d ", root->data);
        preorder(root->leftchild);
        preorder(root->rightchild);
    }
}
// Function to perform postorder(L R V) traversal of the binary
void postorder(treePointer root)
{
    if (root != NULL)
    {
        postorder(root->leftchild);
        postorder(root->rightchild);
        printf("%d ", root->data);
    }
}
/* Function to search a key in the BST */
void search(treePointer root, int key)
{
    treePointer temp ;
    temp= root;
    while (temp != NULL)
    {
```

```
        if (key == temp->data) // Compare the node data with key
        {
            printf("Key found!\n");
            return;
        }
        else if (key < temp->data) // If key less the node data search left
subtree
        {
            temp = temp->leftchild;
        }
        else  // If key greater than the node data search right subtree
        {
            temp = temp->rightchild;
        }
    }
}
O/P:
```

Course Outcomes

1. **Demonstrate Understanding of BST Structure and Properties**:
   - Students will be able to explain the structure and properties of Binary Search Trees, including how elements are organized and how this structure supports efficient search operations.
2. **Implement BST Operations in C**:
   - Students will be able to write a C program that efficiently creates a BST, inserts elements, searches for elements, and traverses the tree using Inorder, Preorder, and Postorder methods.
3. **Perform Tree Traversals**:
   - Students will be able to perform and implement different tree traversal techniques (Inorder, Preorder, Postorder) on a BST, understanding the specific order in which nodes are visited in each traversal.
4. **Search and Manage Data Using BSTs**:
   - Students will be able to search for a given element (KEY) in a BST and report whether it is found, demonstrating their ability to implement and apply search algorithms in a tree structure.

```
/*
10.Develop a Program in C for the following operations on Graph(G) of Cities
a. Create a Graph of N cities using Adjacency Matrix.
b. Print all the nodes reachable from a given starting node in a digraph using
DFS/BFS  method */
```

Course Objectives

1. **Understanding Graph Data Structures**:
   - Students will learn the fundamental concepts of graphs, including the representation of graphs using adjacency matrices, and how to model real-world problems like city connectivity.
2. **Mastering Graph Representations**:
   - Students will gain proficiency in representing graphs using adjacency matrices, understanding how this representation supports efficient operations for graph algorithms.
3. **Implementing Graph Traversal Algorithms**:
   - Students will learn and implement graph traversal algorithms, specifically Depth-First Search (DFS) and Breadth-First Search (BFS), to explore and analyze nodes and edges in a graph.

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

#define MAX 10

int graph[MAX][MAX];
int visited[MAX];
int queue[MAX];
int front = -1, rear = -1;
// Create a graph using Adjacency Matrix values..
void createGraph(int n)
{
    int i, j;
    printf("Enter the adjacency matrix:\n");
    for (i = 0; i < n; i++)
    {
        printf("Enter row %d: ", i + 1);
        for (j = 0; j < n; j++)
        {
            // Enter 0 if edge not existed otherwise 1
```

```
            scanf("%d", &graph[i][j]);
        }
    }
}
// Recursive version of DFS
void DFS(int start, int n)
{
    int i;
    printf("%d ", start);
    visited[start] = 1; // Put 1 for strating vertex that is visited
    for (i = 0; i < n; i++)
    {
        // if vertex is not visted and check if there is edge btween
        // Current vertex to this vertex
        if (!visited[i] && graph[start][i] == 1)
        {
            DFS(i, n); // Call DFS recursively to reach next vertex
        }
    }
}
// BFS Implementation using queue ...
void BFS(int start, int n)
{
    int i, vertex;
    printf("%d ", start);
    visited[start] = 1;    // Put Starting vertex visited
    queue[++rear] = start; // Insert into queue visited first vertex
    while (front <= rear)
    {
        vertex = queue[front++]; // Pop the vertex visited
        for (i = 0; i < n; i++)
        {
            // if vertex is not visted and check if there is edge btween
            // Current vertex to this vertex
            if (!visited[i] && graph[vertex][i] == 1)
            {
```

```
                printf("%d ", i);
                visited[i] = 1;
                queue[++rear] = i; // Insert all the vertices connected to Current
vertex
            }
        }
    }
}
int main()
{
    int ch,i;
    int n, start;
    while (1)
    {
        printf("\n1.Create a Graph\n.2.DFS \n 3.BFS \n 4.Exit \n Enter your
choice:\n");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
            printf("Enter the number of cities: ");
            scanf("%d", &n);
            createGraph(n);
            break;
        case 2:
            printf("\nEnter the starting node: ");
            scanf("%d", &start);
            printf("\nNodes reachable from node %d using DFS: ", start);
            DFS(start, n);
            // Reset visited array for BFS
            for (i = 0; i < n; i++)
            {
                visited[i] = 0;
            }
            break;
        case 3:front = rear = -1; // Reset queue for BFS
```

```
                printf("\nNodes reachable from node %d using BFS: ", start);
                BFS(start, n);
            // Reset visited array for BFS
            for (i = 0; i < n; i++)
            {
                visited[i] = 0;
            }

            printf("\n");
        case 4:exit(0);
            break;
        }
    }
   return 0;
}
```

O/P:

Course Outcomes

1. **Demonstrate Understanding of Graph Concepts**:
   - Students will be able to explain the structure and properties of graphs, including the use of adjacency matrices to represent graphs and the significance of nodes and edges.
2. **Implement Graph Representation Using Adjacency Matrix**:
   - Students will be able to write a C program that creates and initializes a graph of cities using an adjacency matrix, accurately representing the connections between cities.
3. **Perform Graph Traversal with DFS/BFS**:
   - Students will be able to implement Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms to explore a graph, printing all reachable nodes from a given starting node.
4. **Apply Graph Algorithms to Real-World Problems**:
   - Students will be able to use graph traversal techniques to solve practical problems related to city connectivity, demonstrating their ability to model and analyze real-world scenarios using graphs.

/*
11.Given a File of N employee records with a set K of Keys (4-digit) which uniquely determine the records in file F. Assume that file F is maintained in memory by a Hash Table (HT) of m memory locations with L as the set of memory addresses (2-digit) of locations in HT. Let the keys in K and addresses in L are Integers.
Develop a Program in C that uses Hash function H:
K →L as H(K)=K mod m (remainder method), and implement hashing technique to map a given key K to the address space L. Resolve the collision (if any) using linear probing. */

Course Objectives

1. **Understanding Hashing Concepts**:
   - Students will learn the fundamental concepts of hashing, including hash functions, hash tables, and collision resolution techniques, with a focus on the remainder method and linear probing.
2. **Implementing Hash Functions**:
   - Students will gain proficiency in designing and implementing hash functions to map keys to addresses in a hash table, using modulo operations to determine the index.
3. **Managing Collisions with Linear Probing**:
   - Students will learn how to handle collisions in a hash table using linear probing, including techniques for finding the next available slot when a collision occurs.

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_EMPLOYEES 5
#define HT_SIZE 10
// Employee details need to add to Hash table
struct Employee
{
    int key;
    char name[30];
};
// Create an employee hash table..
struct EmployeeHashTable
{
    struct Employee *employees[MAX_EMPLOYEES];
};
// Using hash function division method
```

```
int hash(int key, int m)
{
    return key % m;
}
// insert employee into hash table
void insert(struct EmployeeHashTable *ht, struct Employee *emp, int m)
{
    int index = hash(emp->key, m); // Find index to insert empin hash table
    while (ht->employees[index] != NULL)
    {
        index = (index + 1) % m; //Linear probing finding next position to insert
    }
    ht->employees[index] = emp;
}
// Display the hash table of employees...
void display(struct EmployeeHashTable *ht, int m)
{
    int i;
    printf("Hash Table:\n");
    for (i = 0; i < m; i++)
    {
        if (ht->employees[i] != NULL)
        {
         printf("Index %d: Key=%d, Name=%s\n", i, ht->employees[i]->key,
ht->employees[i]->name);
        }
        else
        {
            printf("Index %d: Empty\n", i);
        }
    }
}
int main()
{
    int i,m;
    struct EmployeeHashTable ht;
```

```
    // Make all employee pointers NULL
    for ( i = 0; i < MAX_EMPLOYEES; i++)
    {
        ht.employees[i] = NULL;
    }
     m = HT_SIZE;
      // Create 4 emps key,values to insert into hash table
    struct Employee e1 = {1000, "Ram"};

    struct Employee e2 = {1001, "Naga"};

    struct Employee e3 = {1002, "Lakshmi"};
    struct Employee e4 = {2002, "Sontosh"};
    insert(&ht, &e1, m);
    insert(&ht, &e2, m);
    insert(&ht, &e3, m);
    insert(&ht, &e4, m);
    display(&ht, m);

    return 0;
}
```

O/P:

Course Outcomes

1. **Demonstrate Understanding of Hashing and Hash Tables**:
   - Students will be able to explain the concepts of hashing, hash functions, and hash tables, including how hash functions are used to map keys to indices in a table.
2. **Implement a Hash Function Using Modulo Operation**:
   - Students will be able to write a C program that implements a hash function H(K)=Kmod mH(K) = K \mod mH(K)=Kmodm to map keys to hash table locations, accurately determining the index for each key.
3. **Handle Collisions Using Linear Probing**:
   - Students will be able to implement collision resolution using linear probing, managing collisions by finding and utilizing the next available slot in the hash table.
4. **Manage Employee Records with Hash Tables**:
   - Students will be able to use hash tables to store and manage employee records, demonstrating their ability to perform insertion, search, and deletion operations efficiently.