MCA Semester II

Data Structures with Algorithms Laboratory

Course Code MMCL207

1. **Implement a Program in C for converting an Infix Expression to Postfix Expression.**

```
#include<stdio.h>
#include<string.h>
int inputpre(char sym) //Function for input precedence
{
switch(sym)
{
case '+' :
case '-' : return 1;
case '*' :
case '/' : return 3;
case '^' :
case '$' : return 6;
case '(' : return 9;
case ')' : return 0;
default : return 7;
}
}
int stackpre(char sym) //Function for stack precedence
{
switch(sym)
{
case '+' :
case '-' : return 2;
case '*' :
case '/' : return 4;
```

```
case '^' :
case '$' : return 5;
case '(' : return 0;
case '#' : return -1;
default : return 8;
}
}
void push (char item, int *top, char s[])
{
s[++(*top)] = item;
}
char pop(int *top, char s[])
{
return s[(*top)--];
}
void infix_to_postfix (char ifix[], char pfix[])
{
int top = -1, i, j = 0;
char s[30] , sym;
push('#',&top,s);
for(i=0;i < strlen(ifix);i++)
{
sym = ifix[i];
while (stackpre(s[top]) >inputpre(sym))
pfix[j++] = pop(&top,s);
if(stackpre(s[top]) != inputpre(sym))
push(sym,&top,s);
else
pop(&top,s);
}
```

```
while(s[top] != '#')
pfix[j++] = pop(&top,s);
pfix[j] = '\0';
}
int main()
{
char ifix[20], pfix[20];
printf("Enter valid infix expression\n");
scanf("%s", ifix);
infix_to_postfix (ifix,pfix);
printf("The postfix expression is = %s", pfix);
return 0;
 }
```

Tracing:

infix expression: `A + B * C`

Initial Setup:

- Input expression (ifix): `"A+B*C"`

- Stack (s): Initially contains `'#'` (with top = 0)

- Output (pfix): Empty string

- Precedence functions:

  - inputpre(): `+,-`=1, `*,/`=3, `^,$`=6, `(`=9, `)`=0, operands=7

  - stackpre(): `+,-`=2, `*,/`=4, `^,$`=5, `(`=0, `#`=-1, operands=8

Tracing Step-by-Step:

1. First character 'A' (i=0):

   - sym = 'A'

   - inputpre('A') = 7

   - stackpre('#') = -1

   - While loop condition: -1 > 7? False

- stackpre('#') != inputpre('A')? -1 != 7 → True → push('A', &top, s)
- Stack: ['#', 'A'], top=1
- pfix: ""

2. Second character '+' (i=1):
   - sym = '+'
   - inputpre('+') = 1
   - stackpre('A') = 8
   - While loop condition: 8 > 1? True → pop 'A' to pfix
     - pfix: "A"
     - Stack: ['#'], top=0
   - Now stackpre('#') = -1 > 1? False
   - stackpre('#') != inputpre('+')? -1 != 1 → True → push('+', &top, s)
   - Stack: ['#', '+'], top=1
   - pfix: "A"

3. Third character 'B' (i=2):
   - sym = 'B'
   - inputpre('B') = 7
   - stackpre('+') = 2
   - While loop condition: 2 > 7? False
   - stackpre('+') != inputpre('B')? 2 != 7 → True → push('B', &top, s)
   - Stack: ['#', '+', 'B'], top=2
   - pfix: "A"

4. Fourth character '*' (i=3):
   - sym = '*'
   - inputpre('*') = 3
   - stackpre('B') = 8
   - While loop condition: 8 > 3? True → pop 'B' to pfix

- pfix: "AB"
- Stack: ['#', '+'], top=1
- Now stackpre('+') = 2 > 3? False
- stackpre('+') != inputpre('*')? 2 != 3 → True → push('*', &top, s)
- Stack: ['#', '+', '*'], top=2
- pfix: "AB"

5. Fifth character 'C' (i=4):
- sym = 'C'
- inputpre('C') = 7
- stackpre('*') = 4
- While loop condition: 4 > 7? False
- stackpre('*') != inputpre('C')? 4 != 7 → True → push('C', &top, s)
- Stack: ['#', '+', '*', 'C'], top=3
- pfix: "AB"

6. End of string (i=5):

- Now we exit the for loop and enter the final while loop to pop remaining stack elements until '#':
- Stack: ['#', '+', '*', 'C']   j loop i=0 to 3 [j=c,*,+,#]
- pop 'C' → pfix: "ABC"
- pop '*' → pfix: "ABC*"
- pop '+' → pfix: "ABC*+"
- Now s[top] = '#', so we stop.
- Final pfix: "ABC*+"

Final Output:

The postfix expression is: `ABC*+`

2. **Design, develop, and execute a program in C to evaluate a valid postfix expression using stack. Assume that the postfix expression is read as a single line consisting of non-negative single digit operands and binary arithmetic operators. The arithmetic operators are + (add), - (subtract), * (multiply) and / (divide).**

```
#include <stdio.h>
#include <ctype.h> // for isdigit()
#define SIZE 100
int stack[SIZE];
int top = -1;
// Push operation
void push(int value) {
    if (top == SIZE - 1) {
        printf("Stack Overflow\n");
    } else {
        stack[++top] = value;
    }
}
// Pop operation
int pop() {
    if (top == -1) {
        printf("Stack Underflow\n");
        return -1;
    } else {
        return stack[top--];
    }
}
int main() {
    char postfix[SIZE];
    int i = 0, op1, op2, result;
    char ch;
    printf("Enter a valid postfix expression (single-digit operands only): ");
    scanf("%s", postfix);  // Read postfix expression as a string
    while ((ch = postfix[i++]) != '\0')
 {
        if (isdigit(ch))
         {
            // Convert char to int and push
            push(ch - '0');
        }
 else
{
            // It's an operator: pop two operands
            op2 = pop();
            op1 = pop();
            switch (ch)
```

```
{
            case '+':
                result = op1 + op2;
                break;
            case '-':
                result = op1 - op2;
                break;
            case '*':
                result = op1 * op2;
                break;
            case '/':
                result = op1 / op2;
                break;
            default:
                printf("Invalid operator: %c\n", ch);
                return 1;
          }
          push(result);
       }
    }
    // Final result will be at the top of the stack
    result = pop();
    printf("Result of the postfix expression: %d\n", result);

    return 0;
}
```

Tracing:

i	ch = postfix[i++]	isdigit(ch)?	Stack Before	Operation Done	Stack After
0	'5'	✅ Yes	[]	push(5)	[5]
1	'3'	✅ Yes	[5]	push(3)	[5, 3]
2	'*'	❌ No	[5, 3]	pop(3), pop(5), 5 * 3 = 15 → push	[15]
3	'2'	✅ Yes	[15]	push(2)	[15, 2]
4	'+'	❌ No	[15, 2]	pop(2), pop(15), 15 + 2 = 17 → push	[17]
5	'\0' (end of input)	loop ends	[17]		

3. Design, develop, and execute a program in C to simulate the working of a queue of integers using an array. Provide the following operations: a. Insert b. Delete c. Display

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 5
int queue [MAX_SIZE];
int front=-1;
int rear= -1;
void isEmpty()
{
    if(rear==-1 && front==-1)
    {
        printf("Queue is empty \n");
    }
    else
    {
        printf("queue is not empty \n");
    }
}
void isFull()
{
    if(rear==MAX_SIZE-1)
    {
        printf("Queue is full \n");
    }
    else{
        printf("Queue is not full \n");
    }
}
void peek()
```

```
{
   if(front==-1 && rear==-1)
   {
      printf("There is no element inside the queue to display");
   }
   else
   {
      printf("The element at the front node is: %d",queue[front]);
   }
}
void Enqueue()
{
   int item;
   if(rear==MAX_SIZE-1)
   {
      printf("Overflow Error");
   }
   else
   {
      if(front==-1)
      {
         front=0;
      }
      printf("Enter the element for Insertion");
      scanf("%d",&item);
      rear++;
      queue[rear]=item;
   }
}
void Dequeue()
```

```
{
   if(front==-1 && rear==-1)
   {
      printf(" queue underflow ");
   }
   else if(front==rear)
   {
      front=rear=-1;
   }
   else
   {
      printf("The deleted element from the queue is",queue[front]);
      front++;
   }
}
void display()
{
   if(front==-1)
   {
      printf("queue is empty");
   }
   else
   {
      printf("Queue elements are:\n");
      for(int i=front;i<=rear;i++)
      printf("%d\n",queue[i]);
   }
}
int main()
{
```

```
int ch;
while (1)
{
 printf("\n 1.sEmpty Operation\n 2.isFull Operation\n 3. peek Operations\n 4.Enqueue Operation\n 5.Dequeue Operation\n 6.Display the Queue\n 7.Exit\n");
 printf("Enter your choice of operations : ");
 scanf("%d", &ch);
   switch (ch)
   {
      case 1:
            isEmpty();
            break;
      case 2:
            isFull();
            break;
      case 3:
            peek();
            break;
      case 4:
            Enqueue();
            break;
      case 5:
            Dequeue();
            break;
      case 6:
            display();
            break;
      case 7:
            exit(0);
      default:
            printf("Incorrect choice \n");
```

```
        }
    }
    return 0;
}
```

4. **Write a C program to simulate the working of a singly linked list providing the following operations: a. Display & Insert b. Delete from the beginning/end c. Delete a given element**

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
// Node structure definition
struct Node {
    int data;
    struct Node *next;
};
// Head pointer initialized to NULL
struct Node *head = NULL;
// Function declarations
void insertAtBeginning(int);
void insertAtEnd(int);
void insertBetween(int, int, int);
void display();
void removeBeginning();
void removeEnd();
void removeSpecific(int);
void main() {
    int choice, value, choice1, loc1, loc2;
    while (1) {
  printf("\n\n****** MENU ******\n1. Insert\n2. Display\n3. Delete\n4. Exit\nEnter your choice: ");
  scanf("%d", &choice);
```

```
switch (choice) {
    case 1:
        printf("Enter the value to be inserted: ");
        scanf("%d", &value);
        printf("Where do you want to insert?\n1. At Beginning\n2. At End\n3. Between\nEnter your choice: ");
        scanf("%d", &choice1);
        switch (choice1) {
            case 1:
                insertAtBeginning(value);
                break;
            case 2:
                insertAtEnd(value);
                break;
            case 3:
                printf("Enter the two values (existing consecutive nodes) where you want to insert between: ");
                scanf("%d%d", &loc1, &loc2);
                insertBetween(value, loc1, loc2);
                break;
            default:
                printf("\nWrong Input!! Try again!!!\n\n");
                break;
        }
        break;
    case 2:
        display();
        break;
    case 3:
        printf("How do you want to delete?\n1. From Beginning\n2. From End\n3. Specific\nEnter your choice: ");
```

```
            scanf("%d", &choice1);
            switch (choice1) {
              case 1:
                  removeBeginning();
                  break;
              case 2:
                  removeEnd();
                  break;
              case 3:
                  printf("Enter the value you want to delete: ");
                  scanf("%d", &loc2);
                  removeSpecific(loc2);
                  break;
              default:
                  printf("\nWrong Input!! Try again!!!\n\n");
                  break;
            }
            break;
        case 4:
            exit(0);
        default:
          printf("\nWrong input!!! Try again!!\n\n");
      }
    }
    getch();
  }
  void insertAtBeginning(int value) {
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = head;
```

```
    head = newNode;
    printf("\nOne node inserted at beginning!!!\n");
}
void insertAtEnd(int value) {
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if (head == NULL) {
  head = newNode;
    } else {
  struct Node *temp = head;
  while (temp->next != NULL)
      temp = temp->next;
  temp->next = newNode;
    }
    printf("\nOne node inserted at end!!!\n");
}

void insertBetween(int value, int loc1, int loc2) {
    struct Node *temp=head;
    struct Node * newNode;
    if (head == NULL || head->next == NULL) {
  printf("\nInsertion not possible, list has insufficient nodes.\n");
  return;
    }
   //  struct Node *temp = head;
    while (temp->next != NULL && !(temp->data == loc1 && temp->next->data == loc2)) {
  temp = temp->next;
printf("Checking: %d -> %d\n", temp->data, temp->next->data);
    }
```

```
    if (temp->next == NULL) {
   printf("\nSpecified positions not found consecutively!\n");
   return;
    }
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = temp->next;
    temp->next = newNode;
    printf("\nOne node inserted between %d and %d!!!\n", loc1, loc2);
}
void removeBeginning() {
    if (head == NULL) {
        printf("\nList is Empty!!!\n");
    } else {
        struct Node *temp = head;
        head = head->next;
        free(temp);
        printf("\nOne node deleted from beginning!!!\n");
    }
}
void removeEnd() {
    if (head == NULL) {
        printf("\nList is Empty!!!\n");
        return;
    }

    if (head->next == NULL) {
        free(head);
        head = NULL;
    } else {
```

```
        struct Node *temp1 = head;
        struct Node *temp2 = NULL;
        while (temp1->next != NULL) {
          temp2 = temp1;
          temp1 = temp1->next;
        }
        temp2->next = NULL;
        free(temp1);
    }
    printf("\nOne node deleted from end!!!\n");
}
void removeSpecific(int delValue) {
    struct Node *temp1=head, *temp2=NULL;
    if (head == NULL) {
  printf("\nList is Empty!!!\n");
  return;
    }
//    struct Node *temp1 = head, *temp2 = NULL;
    if (head->data == delValue) {
        head = head->next;
        free(temp1);
        printf("\nOne node deleted!!!\n");
        return;
    }
    while (temp1 != NULL && temp1->data != delValue) {
        temp2 = temp1;
        temp1 = temp1->next;
    }
    if (temp1 == NULL) {
        printf("\nGiven node not found in the list!!!\n");
```

```
            return;
        }
        temp2->next = temp1->next;
        free(temp1);
        printf("\nOne node deleted!!!\n");
    }
    void display() {
        if (head == NULL) {
            printf("\nList is Empty\n");
        } else {
            struct Node *temp = head;
            printf("\n\nList elements are: ");
            while (temp != NULL) {
                printf("%d ---> ", temp->data);
                temp = temp->next;
            }
            printf("NULL\n");
        }
    }
```

5. **Write a C program to Implement the following searching techniques a. Linear Search b. Binary Search.**

```
    #include <stdio.h>
    // Linear Search Function
    int linearSearch(int arr[], int n, int key) {
        for (int i = 0; i < n; i++) {
            if (arr[i] == key) {
                return i; // Return index if key is found
            }
        }
        return -1; // Return -1 if key not found
    }
```

```
// Binary Search Function (for sorted arrays)
int binarySearch(int arr[], int size, int key) {
    int low = 0;
    int high = size - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == key)
            return mid; // Element found
        else if (arr[mid] < key)
            low = mid + 1; // Search in right half
        else
            high = mid - 1; // Search in left half
    }
    return -1; // Element not found
}
int main() {
    int size, key;
    // Input size
    printf("Enter size of array: ");
    scanf("%d", &size);
    int arr[size];
    // Input sorted array elements
    printf("Enter array elements in sorted order:\n");
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }
    // Input the element to search
    printf("Enter the element to search: ");
    scanf("%d", &key);
```

```
    // Linear Search
    int indexLinear = linearSearch(arr, size, key);
    if (indexLinear != -1)
        printf("Linear Search: Element found at index %d\n", indexLinear);
    else
        printf("Linear Search: Element not found in the array\n");
    // Binary Search
    int indexBinary = binarySearch(arr, size, key);
    if (indexBinary != -1)
        printf("Binary Search: Element found at index %d\n", indexBinary);
    else
        printf("Binary Search: Element not found in the array\n");
    return 0;
}
```

Tracing:

Linear Search:

key = 8

arr = [2, 4, 6, 8, 10] [i=0,1,2,3,4]

Step 1: i = 0 → arr[0] = 2 → Not equal to 8

Step 2: i = 1 → arr[1] = 4 → Not equal to 8

Step 3: i = 2 → arr[2] = 6 → Not equal to 8

Step 4: i = 3 → arr[3] = 8 → Match found ✅

Result: Linear Search returns index 3

Binary Search:

key = 8

arr = [2, 4, 6, 8, 10]

low = 0, high = 4

Step 1:

mid = (0 + 4) / 2 = 2

arr[mid] = arr[2] = 6 → 6 < 8 → Search right half → low = mid + 1 = 3

Step 2:

mid = (3 + 4) / 2 = 3

arr[mid] = arr[3] = 8 → Match found ✅

Result: Binary Search returns index 3

6. **Write a C program to implement the following sorting algorithms using user defined functions: a. Bubble sort (Ascending order) b. Selection sort (Descending order).**

```
#include <stdio.h>
// Function to perform Bubble Sort in Ascending Order
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        // Last i elements are already in place
        for (int j = 0; j < n - i - 1; j++) {
            // Swap if current element is greater than next
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
// Function to perform Selection Sort in Descending Order
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int maxIndex = i;
        for (int j = i + 1; j < n; j++) {
            // Find the maximum element in the unsorted part
            if (arr[j] > arr[maxIndex]) {
                maxIndex = j;
```

```
            }
        }
        // Swap the found maximum element with the first element
        if (maxIndex != i) {
            int temp = arr[i];
            arr[i] = arr[maxIndex];
            arr[maxIndex] = temp;
        }
    }
}
// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main() {
    int arr1[100], arr2[100], n;
    // Input size of array
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    // Input elements
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr1[i]);
        arr2[i] = arr1[i];  // Copy to second array for selection sort
    }
    // Perform Bubble Sort
    bubbleSort(arr1, n);
```

```
    printf("Array after Bubble Sort (Ascending Order):\n");
    printArray(arr1, n);
    // Perform Selection Sort
    selectionSort(arr2, n);
    printf("Array after Selection Sort (Descending Order):\n");
    printArray(arr2, n);
    return 0;
}
```

Tracing:

arr = [5, 2, 8, 1]

Bubble Sort (Ascending)

Pass 1 (i = 0)

- j = 0: compare 5 and 2 → swap → [2, 5, 8, 1]
- j = 1: compare 5 and 8 → no swap
- j = 2: compare 8 and 1 → swap → [2, 5, 1, 8]

Largest element 8 is now at the end.

Pass 2 (i = 1)

- j = 0: compare 2 and 5 → no swap
- j = 1: compare 5 and 1 → swap → [2, 1, 5, 8]

Second largest 5 is at the correct position.

Pass 3 (i = 2)

- j = 0: compare 2 and 1 → swap → [1, 2, 5, 8]

Now fully sorted in **ascending order**.

Selection Sort (Descending)

Pass 1 (i = 0)

- maxIndex = 0 (5)
- j = 1: 2 < 5 → no change

- j = 2: 8 > 5 → maxIndex = 2
- j = 3: 1 < 8 → no change
- Swap arr[0] with arr[2] → [8, 2, 5, 1]

Pass 2 (i = 1)

- maxIndex = 1 (2)
- j = 2: 5 > 2 → maxIndex = 2
- j = 3: 1 < 5 → no change
- Swap arr[1] with arr[2] → [8, 5, 2, 1]

Pass 3 (i = 2)

- maxIndex = 2 (2)
- j = 3: 1 < 2 → no change
- No swap needed

7. Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm (C programming)

```
#include <stdio.h>
#define MAX 100
// Structure to represent an edge
struct Edge {
    int u, v, weight;
};
// Function to sort edges based on weight using Bubble Sort
void sortEdges(struct Edge edges[], int E) {
    for (int i = 0; i < E - 1; i++) {
        for (int j = 0; j < E - i - 1; j++) {
            if (edges[j].weight > edges[j + 1].weight) {
                struct Edge temp = edges[j];
                edges[j] = edges[j + 1];
```

```
                edges[j + 1] = temp;
            }
        }
    }
}
// Find function with path compression
int findParent(int parent[], int i) {
    if (parent[i] == i)
        return i;
    return parent[i] = findParent(parent, parent[i]);
}
// Union function
void unionSet(int parent[], int x, int y) {
    int setX = findParent(parent, x);
    int setY = findParent(parent, y);
    parent[setX] = setY;
}
int main() {
    int V, E;
    struct Edge edges[MAX];
    printf("Enter number of vertices: ");
    scanf("%d", &V);
    printf("Enter number of edges: ");
    scanf("%d", &E);
    // Input all edges
    printf("Enter edges (u v weight):\n");
    for (int i = 0; i < E; i++) {
        scanf("%d %d %d", &edges[i].u, &edges[i].v, &edges[i].weight);
    }
    // Sort edges by weight
```

```
    sortEdges(edges, E);

    int parent[V];
    for (int i = 0; i < V; i++)
        parent[i] = i;
    int minCost = 0;
    printf("Edges in Minimum Cost Spanning Tree:\n");
    for (int i = 0, count = 0; count < V - 1 && i < E; i++) {
        int u = edges[i].u;
        int v = edges[i].v;
        int w = edges[i].weight;
        int setU = findParent(parent, u);
        int setV = findParent(parent, v);
        // If including this edge doesn't cause a cycle
        if (setU != setV) {
            printf("%d - %d : %d\n", u, v, w);
            minCost += w;
            unionSet(parent, setU, setV);
            count++;
        }
    }
    printf("Minimum Cost of Spanning Tree = %d\n", minCost);
    return 0;
}
```

8. From a given vertex in a weighted connected graph, find shortest paths to other vertices Using Dijkstra's algorithm (C programming).

```
#include <stdio.h>
#define MAX 10
#define INF 999 // Infinity
int parent [MAX];
// Function to find the parent of a node
```

```
int find(int i) {
    while (parent[i] != i)
        i = parent[i];
    return i;
}
// Function to perform union of two subsets
void unionSet(int i, int j) {
    int a = find(i);
    int b = find(j);
    parent[a] = b;
}
// Function to implement Kruskal's Algorithm
void kruskal(int cost[MAX][MAX], int n) {
    int i, j, u, v, min, a, b;
    int ne = 1;  // Number of edges in MST
    int mincost = 0;
    // Initialize parent array
    for (i = 1; i <= n; i++)
        parent[i] = i;
    printf("\nThe edges in the Minimum Cost Spanning Tree are:\n");
    while (ne < n) {
        min = INF;
        // Find the minimum cost edge
        for (i = 1; i <= n; i++) {
            for (j = 1; j <= n; j++) {
                if (cost[i][j] < min) {
                    min = cost[i][j];
                    a = u = i;
                    b = v = j;
                }
```

```
            }
        }
        u = find(u);
        v = find(v);
        if (u != v) {
            printf("%d edge (%d,%d) = %d\n", ne++, a, b, min);
            mincost += min;
            unionSet(u, v);
        }
        cost[a][b] = cost[b][a] = INF;  // Remove edge from matrix
    }
    printf("Minimum cost = %d\n", mincost);
}
int main() {
    int a[MAX][MAX], n, i, j;
    printf("=========================================\n");
    printf(" Find Minimum Cost Spanning Tree using Kruskal Algorithm \n");
    printf("=========================================\n");
    printf("\nEnter the number of vertices in the graph: ");
    scanf("%d", &n);
    if (n <= 0) {
        printf("Enter a valid number of vertices.\n");
        return 1;
    }
    printf("\nEnter the cost adjacency matrix\n");
    printf("(Enter 0 for self loops and 999 if no direct edge):\n");
    for (i = 1; i <= n; i++) {
        for (j = 1; j <= n; j++) {
            scanf("%d", &a[i][j]);
        }
```

```
    }

    kruskal(a, n);
    return 0;
}
```