Stacks and Queues

2A. Aim: Write a program to reverse the given string using a stack.

Algorithm:

Step 1: Initialize a char array to store the string and input the string from the user.
Step 2: Get the length of the string by traversing it and incrementing counter till a \0 is encountered.
Step 3: Allocate memory for a stack to push the characters in the string. Initialize top of stack to -1.
Step 4: Traverse the characters in the string, and push each character to the stack.
Step 5: While pushing the characters in the stack, pre-increment the top of stack variable.
Step 6: After all the characters are pushed to the stack, pop all the characters and display the character until top of stack is equal to -1.
Step 7: End the program.

Program: [In next page]

```c
// Exp. 2A: Reverse String using a Stack
// Author: Pranjal Timsina; 20BDS0392
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// Initialize the stack variable
char *stack;
// Returns if stack is empty
int is_empty(char *top_of_stack) { return (*top_of_stack == -1 ); }
// Returns if stack is full
int is_full(int *top_of_stack, int *size_of_stack) {
    return (*top_of_stack == *size_of_stack-1);
}
// Pop the top value from stack
char pop(char *stack, int *top_of_stack) {
    char return_value = stack[*top_of_stack];
    *top_of_stack = *top_of_stack -1;
    return return_value;
}
// Push new data to stack
void push(char *stack, char *data, int *top_of_stack) {
    stack[++*top_of_stack ] = *data;
}

int main() {
    // initialize and input the string
    char my_string[500];
    scanf("%[^\n]s", &my_string);
    int size = strlen(my_string);
    // allocate memory for the stack
    stack = malloc(size * sizeof(char));
    int top_of_stack = -1;
    // push each character of the string to the stack
    for (int i = 0; i < size; i++)
          push(stack, &my_string[i], &top_of_stack);
    // pop each character from the stack and print it
    while (top_of_stack != -1)
          printf("%c", pop(stack, &top_of_stack));
    free(stack); // free the allocated memory
}
```

Output:

```
pranj@DESKTOP-RPVCPE2 MINGW64 /d/College/2nd Semester/DSA/Lab/Assignment 2
$ ./out.exe
Pranjal Timsina is my name
eman ym si anismiT lajnarP
```

Results:

Thus, the program to reverse the string using a stack is implemented.

2B. Aim: Write a program to implement a stack using a linked list to keep student's details such as name, reg no, age, etc.

Algorithm:

Step 1: Initialize variables to temporarily store the name, registration number and age the user inputs.
Step 2: Initialize the head pointer for the linked list, which will also be the pointer to the top of the stack.
Step 3: Until the user enters 0, ask the user to input data.
Step 4: Store the data that the user entered in a struct consisting of attributes name, registration number, age, and a pointer to the next element in the linked list.
Step 5: Change the head pointer to point at the most recent data and the new head should point to the previous head.
Step 6: Go to step 3.
Step 7: Pop the data from the stack until the head points to a NULL pointer.
Step 8: Display the data that was popped, update the head pointer and free the memory for the data that was popped.
Step 9: End program

Program: [In next page]

```c
// Exp 2B: Stack using a linked list to store student details
// Author: Pranjal Timsina; 20BDS0392
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// Creating a node struct to store details
typedef struct node {
    struct node *next;
    char name[100];
    char reg_no[10];
    int age;
}node;

node *head = NULL; // initilaize head pointer
// adds new data to the stack
void push(char new_name[100], char new_reg_no[10], int new_age) {
    node *new_node;
    new_node = (node *) malloc(sizeof(node));

    strcpy(new_node->name, new_name);
    strcpy(new_node->reg_no,new_reg_no);
    new_node->age = new_age;
    new_node->next = head;
    head = new_node;
}
// pops the data from the stack and prints
void pop() {
    node* current;
    current = head;
    head = head->next;
    printf("%s %s %d\n",
           current->name, current->reg_no, current->age);
    free(current);
}

int main() {
    char temp_name[100];
    char temp_reg[10];
    int temp_age;
    while (1) {
        printf("Enter name: [0 to quit] ");
        scanf("%[^\n]s", &temp_name);
        getchar(); // gets the trailing whitespace
```

```
        if (temp_name[0] == '0') break;

        printf("Enter reg. no: ");
        scanf("%s", &temp_reg);
        getchar(); // gets the trailing whitespace

        printf("Enter age: ");
        scanf("%d", &temp_age);
        getchar(); // gets the trailing whitespace

        push(temp_name, temp_reg, temp_age);
    }

    while (head != NULL) pop();

    return 0;
}
```

Output:

```
pranj@DESKTOP-RPVCPE2 MINGW64 /d/College/2nd Semester/DSA/Lab/Assignment 2
$ ./out.exe
Enter name: [0 to quit] Pranjal Timsina
Enter reg. no: 20BDS0392
Enter age: 18
Enter name: [0 to quit] Pratyush Manandhar
Enter reg. no: 20BCE0286
Enter age: 18
Enter name: [0 to quit] 0
Pratyush Manandhar 20BCE0286 18
Pranjal Timsina 20BDS0392 18
```

Results:

Thus, the program to store student details using a stack implemented using a linked list was written.

2C. Aim: Write and implement a program to convert the given infix expression to a postfix expression.

Algorithm:

Step 1: Input the infix expression in a string variable.
Step 2: Initialize a stack to store the operators. Initialize top of stack to -1.
Step 3: Surround the infix expression with '(' and ')' to reduce complexity.
Step 4: Traverse the infix expression.
Step 5: If a number is encountered, append the number to the postfix expression.
Step 6: If a left parenthesis is encountered, push the left parenthesis to the stack.
Step 7: If a right parenthesis is encountered, pop all the operators until the matching left parenthesis is found and append the operand to the postfix expression.
Step 8: If an operator is found and if the stack is empty push the operand to the stack.
Step 9: If an operator is found and if it has greater precedence than the operator on the top of the stack, push the new operator to the stack.
Step 10: If an operator is found and if it has lower or equal precedence than the operator on the top of the stack, pop all operators until the stack is empty or the operators on top of the stack has greater precedence that the new operator. Append the popped operator to the postfix expression.
Step 11: Display the postfix expression.

Program: [In next page]

```cpp
// Exp 2C: Convert infix expression to postfix
// Author: Pranjal Timsina; 20BDS0392
#include <iostream>
#include <ctype.h>
#include <string>
#define MAX 500

// Initializing a stack
std::string stack[MAX];
int top = -1;

// functions to check if stack is empty or full
bool is_empty() { return (top == -1); }
bool is_full() { return (top == MAX-1); }

// returns the top element of stack without removing
std::string peek() {
    if (is_empty()) { throw "Stack underflow!";}
    return stack[top];
}

// pops the top most element of the stack and decrements top
std::string pop() {
    if (is_empty()) { throw "Stack underflow!";}
    return stack[top--];
}

// pushes data to the stack and increments top
void push(std::string data) {
    if (is_full()) {throw "Stack Overflow!";}
    stack[++top] = data;
}

// if a is an operand, returns its precedence
// if a is not an operand, returns 0
int operand(char a) {
    switch (a) {
        case '+': case '-': return 1;
        case '*': case '/': case '%': return 2;
        case '^': return 3;
        default: return 0;
    }
}
```

```
// if a is a left parentheses returns -1
// if a is a right parentheses returns 1
// if a is not a parentheses returns 0
int parentheses(char a) {
    switch (a) {
        case '{': case '(': case '[': return -1;
        case '}': case ']': case ')': return 1;
        default: return 0;
    }
}

// main function to convert infix to postfix
std::string infix_to_postfix(std::string infix) {
    std::string postfix{""}, temp; // initialize variables
    infix = '(' + infix + ')'; // surround the expression
    // variables for processing the infix expression
    char current; int number;

    // loop to traverse the characters in infix expresison
    for (int i = 0; infix[i] != '\0'; i++){

        current = infix[i]; // storing in variable for ease
        // continues if the current character is whitespace
        if (current == ' ') continue;
        else if (isdigit(current)) {
            // if the current character is a digit, it checks the
             // adjacent characters
            // to see if they are digits too, and gets the full
            // number
            number = (int) (current - '0');
            while (i+1 < MAX
                && infix[i+1] != '\0'
                && isdigit(infix[i+1]))
            {
                i = i+1;
                current = infix[i];
                number = number*10 + (int) current - '0';
            }
```

```
            // adds the number to the postfix expression
            postfix = postfix + std::to_string(number) + " ";
        } else {
            // storing the value of char current to a string
            // to make it a uniform data type for pushing to stack
            temp = current;
            // if the current character is a left parentheses
            // pushes it to the stack
            if (parentheses(current) == -1) {
                push(temp);
            }
            // if the current character is a right parentheses
            // pops everything in the stack and adds it to the
           // postfix expr
            // until a left parentheses is found
            else if (parentheses(current) == 1) {
                while (peek()[0] != '('){
                    postfix +=  pop() + " ";
                }
                // popping the left parentheses
                char temp = pop()[0];
            }
            // if the stack is empty pushes the operand to stack
            // or the precedence of current operand is greater that
             // the precedence of the stack top
            else if (is_empty()
                || operand(current) > operand(peek()[0]))
                push(temp);
            else {
                // pops all operands which have lower precedence
                // than the current operand
                // then pushes the current operand to the stack
```

```
                while(!is_empty()
                     && (operand(current) <= operand(peek()[0]))) {
                    postfix += pop() + " ";
                }
                push(temp);
            }
        }
    }
    return postfix;
}

int main() {
    std::string infix;
    std::cin >> infix;
    std::cout << infix_to_postfix(infix) << std::endl;
    return 0;
}
```

Output:

```
pranj@DESKTOP-RPVCPE2 MINGW64 /d/College/2nd Semester/DSA/Lab/Assignment 2
$ ./out.exe
(5+4)-(5^6)
5 4 + 5 6 ^ -
```

Results:

Thus, the program to change infix to postfix is implemented.

2D. Aim: Write and implement a program to evaluate infix expressions

Algorithm:

Step 1: Input the infix expression in a string variable.
Step 2: Initialize a stack to store the operators. Initialize top of stack to -1.
Step 3: Surround the infix expression with '(' and ')' to reduce complexity.
Step 4: Traverse the infix expression.
Step 5: If a number is encountered, append the number to the postfix expression.
Step 6: If a left parenthesis is encountered, push the left parenthesis to the stack.
Step 7: If a right parenthesis is encountered, pop all the operators until the matching left parenthesis is found and append the operand to the postfix expression.
Step 8: If an operator is found and if the stack is empty push the operand to the stack.
Step 9: If an operator is found and if it has greater precedence than the operator on the top of the stack, push the new operator to the stack.
Step 10: If an operator is found and if it has lower or equal precedence than the operator on the top of the stack, pop all operators until the stack is empty or the operators on top of the stack has greater precedence that the new operator. Append the popped operator to the postfix expression.
Step 11: Create another stack to store the operands. Initialize the top of stack to -1.
Step 12: Traverse the postfix expression from left to right.
Step 13: If a number is encountered, push the number to the stack.
Step 14: If an operator is encountered, pop two numbers from the stack and apply the operator, and push the result back in the stack.
Step 15: Repeat step 13 and 14 until top of stack is -1.
Step 16: Pop and display the last number in the stack.
Step 17: End program.

Program: [In next page]

```
// Exp 2D: Program to evaluate infix expressions
// Author: Pranjal Timsina; 20BDS0392
#include <iostream>
#include <ctype.h>
#include <cmath>
#include <string>
// Initializing a stack
#define MAX 500
std::string stack[MAX];
int top = -1;
// functions to check if stack is empty or full
bool is_empty() { return (top == -1); }
bool is_full() { return (top == MAX-1); }
// returns the top element of stack without removing
std::string peek() {
    return stack[top];
}
// pops the top most element of the stack and decrements top
std::string pop() {
    return stack[top--];
}
// pushes data to the stack and increments top
void push(std::string data) {
    if (is_full()) {throw "Stack Overflow!";}
    stack[++top] = data;
}
// returns precedence of an operator
int operator_precedence(char a) {
    switch (a) {
        case '+': case '-': return 1;
        case '*': case '/': case '%': return 2;
        case '^': return 3;
        default: return 0;
    }
}
// -1 for left, 1 for right parantheses
int parentheses(char a) {
    switch (a) {
        case '{': case '(': case '[': return -1;
        case '}': case ']': case ')': return 1;
        default: return 0;
    }
}
```

```
// function to convert infix to postfix
std::string infix_to_postfix(std::string infix) {
    std::string postfix{""}, temp; // initialize variables
    infix = '(' + infix + ')'; // surround the expressions
    // variables for processing the infix expression
    char current; int number;
    // loop to traverse the characters in infix expresison
    for (int i = 0; infix[i] != '\0'; i++){
        current = infix[i]; // storing in variable for ease
        // continues if the current character is whitespace
       if (current == ' ') {
            continue;
        }
        else if (isdigit(current)) {
            // checks adjacent characters for digits
            number = (int) (current - '0');
            while (i+1 < MAX
                    && infix[i+1] != '\0'
                    && isdigit(infix[i+1]))
           {
                i = i+1;
                current = infix[i];
                number = number*10 + (int) current - '0';
            }
            // adds the full number to the postfix expression
            postfix = postfix + std::to_string(number) + " ";
        } else {
            // storing the value of char current to a string
            // to make it a uniform data type for pushing to stack
            temp = current;

            // if the current character is a left parentheses
            // pushes it to the stack
            if (parentheses(current) == -1) push(temp);
            // if the current character is a right parentheses
            // pops everything in the stack and adds it to the
             // postfix expr until a left parentheses is found
            else if (parentheses(current) == 1) {
                while (peek()[0] != '('){
                    postfix +=  pop() + " ";
                }
                pop(); // popping the left parentheses
            }
```

```
// initialize stack for operators
int int_stack[500];
int int_top = -1;
void int_push(int data) {
    if(int_top != 499) int_stack[++int_top] = data;
}
int int_pop() {
    if (int_top != -1) return int_stack[int_top--];
    else return 1;
}
// takes operators, operators and applies it
int apply_operator(int a, int b, char op) {
    switch (op) {
        case '+':
            return a + b;
        case '-':
            return a - b;
        case '*':
            return a * b;
        case '/':
            return a / b;
        case '^':
            return pow(a,b);
        default:
            return 0;
    }
}

int main() {
    std::string infix;
    std::cin >> infix;
    std::string postfix = infix_to_postfix(infix);
    char current;
    int number;
    // traverses the postfix expression
    for (int i = 0; i < MAX && postfix[i] != '\0'; i++) {
        current = postfix[i];
        if (current == ' ') {
            continue;
        }
```

```
if (isdigit(current)) {
            number = (int) current - '0';
            // if the current char is an number,
            // it checks adjacent chars to find out the complete
            // number
            while (i+1 < MAX && postfix[i+1] != '\0'
                    && isdigit(postfix[i+1]))
            {
                current = postfix[++i];
                number = number*10 + ((int)current - '0');
            }
            int_push(number); // pushing number to the stack
        } else if (operator_precedence(current)) {
            // pops two numbers and applies the operator
            // then pushes back the result to the stack
           int_push(apply_operator(int_pop(), int_pop(), current));
        }
    }
    std::cout << "ans = " << int_pop() << std::endl;
    return 0;
}
```

Output:

```
pranj@DESKTOP-RPVCPE2 MINGW64 /d/College/2nd Semester/DSA/Lab/Assignment 2
$ ./out.exe
2^((6-1)*(1/5))
ans = 1

pranj@DESKTOP-RPVCPE2 MINGW64 /d/College/2nd Semester/DSA/Lab/Assignment 2
$ ./out.exe
8*8-9*9
ans = -17
```

Results:

Thus, the program to evaluate infix expression is implemented.

2E. Aim: Implement a data structure to keep students' details in the same order of their admission, with provision to add and remove in the same of their entry in the list.

Algorithm:

Step 1: Define a struct 'node' with attributes name, registration number, age and the pointer to the next node.
Step 2: Initialize the head and tail of the queue to a NULL pointer.
Step 3: Implement the function enqueue which allocates memory to a new node, and stores the relevant data to the new node. If the head pointer was previously NULL, make the head pointer and the tail pointer to point to the new node. Set the next node attribute of the current new node to be NULL. If the head pointer was not NULL, make the tail of the queue point to the new node, and set the next node attribute of the current new node to be NULL.
Step 4: Implement the function dequeue which sets the value of head to what is pointed by it currently. Print the details of the current head. Then free the memory allocated to the current node.
Step 5: In the main program. While the user does not input 0, ask them to choose to enqueue, dequeue or exit.
Step 6: If the user chooses to enqueue, input the relevant data from the user and enqueue the data.
Step 7: If the user chooses to dequeue, dequeue and display the data pointed by the tail pointer.
Step 8: Go to step 5 until the user chooses to exit.

Program:

```c
// Exp 2E: Insert and remove student's details in the same order
// of their admission
// Author: Pranjal Timsina, 20BDS0392
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 100
// create a struct for storing student data
typedef struct node {
    struct node *next;
    char name[100];
    char reg_no[10];
    int age;
}node;
// initialize head and tail of the queue
node *head = NULL;
node *tail = NULL;
// function to add new data to the end of queue
void enqueue(char new_name[100], char new_reg_no[10], int new_age)
{
    node *new_node;
    new_node = (node *) malloc(sizeof(node));
    strcpy(new_node->name, new_name);
    strcpy(new_node->reg_no,new_reg_no);
    new_node->age = new_age;

    if (head == NULL) {
        head = new_node;
        tail = new_node;
        new_node->next = NULL;
    } else {
        tail->next = new_node;
        tail=new_node;
        tail->next = NULL;
    }
}
// function to remove data from the front queue
```

```
void dequeue() {
    node* current;
    current = head;
    head = head->next;
    printf("%s %s %d\n", current->name, current->reg_no, current-
>age);
    free(current);
}
void menu() {
    printf("1. Add, 2. Remove, 3. Exit\n");
    printf("Exiting will display the whole queue\n");
}
int main() {
    // initialize variables for input
    char temp_name[100];
    char temp_reg[10];
    int temp_age;
    int choice;
    while (1) {
        menu(); // display menu
        scanf("%d",&choice); // get user choice
        getchar();
        switch (choice) {
            case 1:
                printf("Enter name: ");
                scanf("%[^\n]s", &temp_name);
                getchar(); // get trailing whitespace
                printf("Enter reg. no: ");
                scanf("%s", &temp_reg);
                getchar(); // get trailing whitespace

                printf("Enter age: ");
                scanf("%d", &temp_age);
                getchar(); // get trailing whitespace

                enqueue(temp_name, temp_reg, temp_age);
                break;
            case 2:
                if (head != NULL) {
                    dequeue();
                }
                break;
            case 3:
```

```
                while (head != NULL)
                    dequeue();
                return 0;
            default:
                printf("Invalid input\n");
                break;
        }
    }
    return 0;
}
```

Output:

```
pranj@DESKTOP-RPVCPE2 MINGW64 /d/College/2nd Semester/DSA/Lab/Assignment 2
$ ./out.exe
1. Add, 2. Remove, 3. Exit
Exiting will display the whole queue
1
Enter name: Pranjal Timsina
Enter reg. no: 20BDS0392
Enter age: 18
1. Add, 2. Remove, 3. Exit
Exiting will display the whole queue
1
Enter name: John Wick
Enter reg. no: 123
Enter age: 28
1. Add, 2. Remove, 3. Exit
Exiting will display the whole queue
1
Enter name: Aarav Timsina
Enter reg. no: 1288
Enter age: 15
1. Add, 2. Remove, 3. Exit
Exiting will display the whole queue
2
Pranjal Timsina 20BDS0392 18
1. Add, 2. Remove, 3. Exit
Exiting will display the whole queue
3
John Wick  123 28
Aarav Timsina 1288 15
```

Results:

Thus, the program to [do something] is implemented with time complexity $O(sth)$.

2F. Aim: Implement a data structure to keep sales details in the order of their entry. The number of entries is unknown and dynamically changing.

Algorithm:

Step 1: Define a struct ‘node’ with attributes product name, product id, price and the pointer to the next node.
Step 2: Initialize the head and tail of the queue to a NULL pointer.
Step 3: Implement the function enqueue which allocates memory to a new node, and stores the relevant data to the new node. If the head pointer was previously NULL, make the head pointer and the tail pointer to point to the new node. Set the next node attribute of the current new node to be NULL. If the head pointer was not NULL, make the tail of the queue point to the new node, and set the next node attribute of the current new node to be NULL.
Step 4: Implement the function dequeue which sets the value of head to what is pointed by it currently. Print the details of the current head. Then free the memory allocated to the current node.
Step 5: In the main program. While the user does not input 0, ask them to choose to enqueue, dequeue or exit.
Step 6: If the user chooses to enqueue, input the relevant data from the user and enqueue the data.
Step 7: If the user chooses to dequeue, dequeue and display the data pointed by the tail pointer.
Step 8: Go to step 5 until the user chooses to exit.

Program: [In next page]

```c
// Exp 2F: Dynamic sales entries
// Author: Pranjal Timsina; 20BDS0392
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// define a struct node to store sales detail
typedef struct node {
    struct node *next;
    char product_name[100];
    int product_id; int price;
}node;
// initialize head and tail pointers of the queue
node *head = NULL, *tail = NULL;
void enqueue(char new_name[100], int new_product_id, int new_price)
 {
    // create a new node and allocate memory
    node *new_node;
    new_node = (node *) malloc(sizeof(node));
    // copy the relevant contents to the struct
    strcpy(new_node->product_name, new_name);
    new_node->product_id = new_product_id;
    new_node->price = new_price;
    if (head == NULL) {
        // if head is NULL, update HEAD and TAIL
        // to point to the current node
        head = new_node;
        tail = new_node;
        new_node->next = NULL; // set the next to NULL
    } else {
        // update the TAIL to point to next node
        tail->next = new_node;
        tail=new_node;
        tail->next = NULL;
    }
}
void dequeue() {
    // reference the current head
    node* current = head;
    head = head->next; // update head pointer
    printf("%d %s %d\n", current->product_id,
        current->product_name, current->price);
    free(current); // free the memory
}
```

```
int main() {
    // variables to store user input
    char temp_name[100];
    int temp_price;
    int temp_id;
    while (1) {
        printf("Enter product name: [0 to quit] ");
        scanf("%[^\n]s", &temp_name);
        getchar(); // get trailing whitespace
        if (temp_name[0] == '0') {
            break; // condition to exit
        }
        printf("Enter product ID: ");
        scanf("%d", &temp_id);
        getchar(); // get trailing whitespace
        printf("Enter product price: ");
        scanf("%d", &temp_price);
        getchar(); // get trailing whitespace
        // enqueue with the relevant details
        enqueue(temp_name, temp_id, temp_price);
    }
    while (head != NULL){
        // call the dequeue function which prints
        // data of the dequeued item.
        dequeue();
    }
}
```

Output: [In next page]

```
pranj@DESKTOP-RPVCPE2 MINGW64 /d/College/2nd Semester/DSA/Lab/Assignment 2
$ ./sales.exe
Enter product name: [0 to quit] Pen
Enter product ID: 1
Enter product price: 20
Enter product name: [0 to quit] Pencil
Enter product ID: 2
Enter product price: 10
Enter product name: [0 to quit] Book
Enter product ID: 3
Enter product price: 200
Enter product name: [0 to quit] Copy
Enter product ID: 4
Enter product price: 80
Enter product name: [0 to quit] 0
1 Pen 20
2 Pencil 10
3 Book 200
4 Copy 80
```

Results:

Thus, the program to store the details of unknown number of sales entries is implemented in the order they are made is implemented using a queue.

2G. Aim: Implement a circular queue with functionality to add and remove data

Algorithm:

Step 1: Implement function enqueue that takes head, tail, size, data, and an array-based queue as the arguments. If $head = -1$, then it initializes head and tail both to 0 then inserts data at queue [0]. Else if the queue is not full, then it sets $tail = (tail + 1)\%size$, and inserts new data at queue [tail].
Step 2: Implement function dequeue that takes head, tail, size, and the array-based queue as the arguments and returns the element at the head of the queue. Then it sets the $head = (head + 1)\%size$ if there still remain other elements in the queue, else sets it to -1.
Step 3: Implement function display queue that iterates cyclically from head to tail and prints the elements.
Step 4: Input the size of queue from the user and initialize an array of size N.
Step 5: Initialize the head and tail of the queue to -1.
Step 6: While the user does not enter 0, display the menu, and then add/remove/display data as per the user's input.
Step 7: End program.

Program: [In next page]

```
// Exp 2G: Implement a circular queue
// Author: Pranjal Timsina; 20BDS0392
#include <iostream>

void enqueue(int &head, int &tail, int &size, int queue[],
             int &data)
{
    if (head == -1) { // initialize head and tail if queue is empty
        head = 0; tail = 0;
        queue[tail] = data;
    } else if (!((head == 0 && tail == size-1)
                    ||(tail == head-1)))
    {
            // increment tail and use % for making it cyclic
            tail = (tail+1)%size;
            queue[tail] = data;
    } else std::cerr << "Overflow!!\n"; // give error message
}

int dequeue(int &head, int &tail, int size, int queue[]) {
    if (head != -1) { // check if queue is not empty
        int return_value{queue[head]};
        if (head == tail) {
            // if one element it queue, resets head and tail
            tail = -1; head = -1;
        } else {
          // increment head, use % for making it cyclic
            head = (head+1)%size;
        }
        return return_value;
    } else std::cerr << "Underflow, returning garbage value!\n";
          // throw error
}
```

[Continued in next page]

```
void display_queue(int &head, int &tail, int &size, int queue[]) {
    if (head != -1) {
       std::cout << "--------------Queue--------------\n Head -> ";
       int i = head; // start at head of queue
       while (true) {
            if (i == tail) {
                std::cout << queue[i];
                break; // if tail is reached, break
            } else {
                std::cout << queue[i] << " "; // print element
            }
           i = (i + 1)%size; // increment i cyclically
       }
        std::cout << " <- Tail\n-------------------------------\n";
    } else {
        std::cerr << "empty queue!\n"; // throw error
    }
}
int main() {
    int size, choice, head{-1}, tail{-1};
    std::cout << "Enter size of queue: ";
    std::cin >> size;
    int queue[size]; // initialize queue
    while (true) {
        std::cout << "1. Display 2. Enqueue 3. Dequeue 0. Exit\n";
        std::cin >> choice;
        switch (choice) { // handle user choice
        case 0: return 0; // exit program
        case 1:
            display_queue(head, tail, size, queue);
            continue;
        case 2:
            std::cout << "Data: ";
            std::cin >> choice; // get user input
            enqueue(head, tail, size, queue, choice);
            continue;
        case 3:
            std::cout << dequeue(head, tail, size, queue)
                      << " dequeued\n";
        default: continue;
        }
    }
}
```

Output:

```
pranj@DESKTOP-RPVCPE2 MINGW64 /d/College/2nd Semester/DSA/Lab/Assignment 2
$ ./circular_queue.exe
Enter size of queue: 3
1. Display 2. Enqueue 3. Dequeue 0. Exit
2
Enter data: 3
1. Display 2. Enqueue 3. Dequeue 0. Exit
2
Enter data: 4
1. Display 2. Enqueue 3. Dequeue 0. Exit
2
Enter data: 5
1. Display 2. Enqueue 3. Dequeue 0. Exit
1
--------------Queue--------------
 Head -> 3 4 5 <- Tail
---------------------------------
1. Display 2. Enqueue 3. Dequeue 0. Exit
3
3 was dequeued
1. Display 2. Enqueue 3. Dequeue 0. Exit
3
4 was dequeued
1. Display 2. Enqueue 3. Dequeue 0. Exit
1
--------------Queue--------------
 Head -> 5 <- Tail
---------------------------------
1. Display 2. Enqueue 3. Dequeue 0. Exit
2
Enter data: 9
1. Display 2. Enqueue 3. Dequeue 0. Exit
1
--------------Queue--------------
 Head -> 5 9 <- Tail
---------------------------------
1. Display 2. Enqueue 3. Dequeue 0. Exit
0
```

Results:

Thus, a circular queue is implemented.

2H. Aim: Implement a solution to Josephus problem using a circular linked list to identify and print the id of the winner. Users can get the number of players and the id of the starter.

Algorithm:

Step 1: Input number of people ‘n’ and the skip value ‘m’.
Step 2: Initialize a linked list with n people.
Step 3: Traverse linked list removing every m^{th} person until a single person is left
Step 4: Print the position of the remaining person and declare it as winner.

Program: [In next page]

```c
// Exp2H: Solution to Josephus problem
// Author: 20BDS0392; Pranjal Timsina

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

// define struct Node to store each person
typedef struct Node {
    int data; // data stores the index of the person
    struct node *next;
} Node;

Node *newPerson(int data) { // func to create a newPerson
    Node *new = (Node* ) malloc(sizeof(Node));
    new->next = new;
    new->data = data;
    return new;
}

// main function to solve the problem
void solve(int m, int n) {
    Node *start = newPerson(1); // starting person
    Node *prev = start;

    // loop to create n people
    for (int i = 2; i <= n; i++) {
        prev->next = newPerson(i);
        prev = prev->next;
    }
    prev->next = start; // make the linked list circular
    Node *winner = start, *temp = start; // initizalize two ptrs
    // loop runs while there are multiple people
```

```
  while (winner->next != winner)
    {
        // traverse m people
        int count = 1;
        while (count != m)
        {
            temp = winner;
            winner = winner->next;
            count++;
        }
        // prepare kill the mth person
        temp->next = winner->next;
        free(winner); // kill the mth person
        winner = temp->next; // put the next pointer to ptr1
    }
    printf ("Josephus Position is %d", winner->data);
    free(winner);
}


int main() {
    int people, skip;
    printf("Enter number of people and skip value: ");
    scanf("%d%d", &people, &skip);
    solve(skip, people);
    return 0;
}
```

Output:

```
PS D:\DSA-Assignments\Assignment 2> .\out.exe
Enter number of people and skip value: 9000 55
Josephus Position is 7081

PS D:\DSA-Assignments\Assignment 2> .\out.exe
Enter number of people and skip value: 12311 8
Josephus Position is 11989
```

Results:

Thus, the Josephus problem is solved using a circular linked list.

2I. Aim: Implement the Tower of Hanoi problem using recursion. User can give the number of disks; print each step of disk movement.

Algorithm:

Step 1: Input the number of discs
Step 2: Implement a recursive function shift described below.
Step 3: The shift function takes in arguments: number of discs, from tower, to tower, and auxiliary tower.
Step 4: If the number of discs is 1 display Move N discs from *'from tower'* to *'to tower'*.
Step 5: Else shift n-1 discs from tower *'from'* to tower *'aux'* using tower *'to'* as an auxiliary tower. Then display Move N discs from *'from tower'* to *'to tower'*.
Step 6: Shift n-1 discs from tower *'aux'* to tower *'to'*.
Step 7: End program.

Program:

```c
// Exp 2G: Solution to towers of Hanoi
// Author: Pranjal Timsina; 20BDS0392
#include <stdio.h>

//  Moves n discs from tower 'from' to tower 'to' using tower 'aux'
void shift (int discs, int from, int to, int aux) {
    if (discs == 1) // base case for recursion
        printf ("Move  disc %d from %d to  %d\n", discs, from, to);
    else {
     // shift n-1 discs to 'aux' using 'to' as helper
        shift (discs-1, from, aux, to);
        printf ("Move  disc %d from %d to  %d\n", discs, from, to);
     // shift n-1 discs from 'aux' to 'to' using 'from' as helper
        shift (discs-1, aux, to, from);
    }
}

int main() {
    int discs;
    printf("Enter discs: "); scanf("%d", &discs);
    // move discs from tower 1 to 3 using 2 as helper
    shift(discs, 1, 3, 2);
}
```

Output: [In next page]

```
pranj@DESKTOP-RPVCPE2 MINGW64 /d/College/2nd Semester/DSA/Lab/Assignment 2
$ out.exe/
Enter discs: 3
Move  disc 1 from 1 to  3
Move  disc 2 from 1 to  2
Move  disc 1 from 3 to  2
Move  disc 3 from 1 to  3
Move  disc 1 from 2 to  1
Move  disc 2 from 2 to  3
Move  disc 1 from 1 to  3
```

Results:

Thus, a solution to the Towers of Hanoi problem is implemented.

2J. Aim: Implement a program to perform polynomial addition using a linked list to represent polynomials.

Algorithm:

Step 1: Initialize 3 linked lists for storing the 1st, 2nd and the resulting polynomial.
Step 2: Input 2 polynomials from the user.
Step 3: Parse the input and store each degree of the polynomial as a node in the linked list.
Step 4: Traverse the linked lists while both are not null.
Step 5: If the nodes of both the lists are of the same degree, add them and store the sum. Traverse to the next node in both the polynomials.
Step 6: If the node of one of the polynomials is greater add that to the polynomial to the result and traverse the polynomial.
Step 7: If any of the polynomials still have nodes left, append it to the result.
Step 8: Display the resulting polynomial.

Program: [In next page]

```c
// Exp 2J: Polynomial Addition
// Author: Pranjal Timsina; 20BDS0392
#include<stdio.h>
#include<stdlib.h>

struct Node { // def node to store data
    int coeff;
    int pow;
    struct Node* next;
};

void inputPoly(struct Node** poly) {
    int coeff, exp, cont; // temp variables for input

    struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
    *poly = temp;

    do { // ensures loop runs at least once
        printf("Enter the coeffecient and exponent (eg 3 2): ");
        scanf("%d%d", &coeff, &exp);
        temp->coeff = coeff;
        temp->pow = exp;
        temp-> next = NULL; // set next pointer to null
        printf("Enter 0 to terminate, anything else to continue: ");
        scanf("%d", &cont);
        if (cont) { // create new node if more data
            temp->next = (struct Node*)malloc(sizeof(struct Node));
            temp = temp->next;
            temp->next = NULL; // point the new node to null
        }
    } while(cont);
}
```

```
void printPoly(struct Node* poly) {
    while (poly != NULL) { // terminate if pointer is null
        printf("%dx^%d ", poly->coeff, poly->pow);
        poly = poly->next; // traverse
        if(poly != NULL) printf(" + ");
    }
    printf("\n");
}

void addPolynomials(struct Node** result, struct Node* first, struct N
ode* second) {
    struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
    temp->next = NULL; // create node to store result
    *result = temp;
    // loop while the 2 polys are not null
    while (first && second) {
        if(first->pow > second->pow) {
            // add the first node as it is
            temp->coeff = first->coeff;
            temp->pow = first->pow;
            first = first->next;
        } else if(first->pow < second->pow) {
            // add the second node as it is
            temp->coeff = second->coeff;
            temp->pow = second->pow;
            second = second->next;
        } else {
            // add the sum of the two polys
            temp->coeff = first->coeff + second->coeff;
            temp->pow = first->pow;
            first = first->next;
            second = second->next;
        }

        if (first && second) {
            // create another node for result
```

```
            temp->next = (struct Node*)malloc(sizeof(struct Node));
            temp = temp->next;
            temp->next = NULL;
        }
    }
    // if any of the polynomials contain nodes, add them
    while (first || second) {
        temp->next = (struct Node*)malloc(sizeof(struct Node));
        temp = temp->next;
        temp->next = NULL;

        if (first) {  // add the remaining elems of 1st poly
            temp->coeff = first->coeff;
            temp->pow = first->pow;
            first = first->next;
        } else if(second) { // add the remaining elemes of 2nd poly
            temp->coeff = second->coeff;
            temp->pow = second->pow;
            second = second->next;
        }
    }
}

int main() {
    // create nodes
    struct Node* first = NULL;
    struct Node* second = NULL;
    struct Node* result = NULL;
    // input data
    printf("Enter the corresponding data:-\n");
    printf("First polynomial:\n");
    inputPoly(&first);
    printPoly(first);
    printf("Second polynomial:\n");
    inputPoly(&second);
```

```
    printf("\n The First polynomail is: ");
    printPoly(first);
    printf("The Second polynomial is: ");
    printPoly(second);
    // add the polynomials
    addPolynomials(&result, first, second);
    printf("The Resulting polynomial is: ");
    // print results
    printPoly(result);
    return 0;
}
```

Output:

```
PS D:\DSA-Assignments\Assignment 2> .\out.exe
Enter the corresponding data:-
First polynomial:
Enter the coeffecient and exponent (eg 3 2): 3 2
Enter 0 to terminate, anything else to continue: 1
Enter the coeffecient and exponent (eg 3 2): 2 1
Enter 0 to terminate, anything else to continue: 1
Enter the coeffecient and exponent (eg 3 2): 5 0
Enter 0 to terminate, anything else to continue: 0
3x^2  + 2x^1  + 5x^0
Second polynomial:
Enter the coeffecient and exponent (eg 3 2): 4 3
Enter 0 to terminate, anything else to continue: 1
Enter the coeffecient and exponent (eg 3 2): 5 2
Enter 0 to terminate, anything else to continue: 1
Enter the coeffecient and exponent (eg 3 2): 6 0
Enter 0 to terminate, anything else to continue: 0

 The First polynomail is: 3x^2  + 2x^1  + 5x^0
The Second polynomial is: 4x^3  + 5x^2  + 6x^0
The Resulting polynomial is: 4x^3  + 8x^2  + 2x^1  + 11x^0
```

Results:

Thus, a program to add polynomials is implemented using a linked list.