Name: **Date:**

Class: MCA-II **Roll No.:**

Subject: Data Structure

Experiment No. 1

Aim: Implement a rational number as a new data type.

Solution:

```
#include <stdio.h>
#include <stdlib.h>
        typedef struct {
                int numerator;
                int denominator;
            } Rational;
                int gc(int a, int b) {
                if (b == 0) {
                return a;
            }   return gcd(b, a % b);
       }
             Rational simplify_rational(Rational r) {
             if (r.denominator == 0) {
              fprintf(stderr, "Error: Denominator cannot be zero.\n");
             exit(EXIT_FAILURE);
      }
         if (r.numerator == 0) {
          r.denominator = 1;
          return r;
  }
      int commonDivisor = gcd(abs(r.numerator), abs(r.denominator));
      r.numerator /= commonDivisor;
     r.denominator /= commonDivisor;
     if (r.denominator < 0) {
     r.numerator = -r.numerator;
     r.denominator = -r.denominator;
}  return r;
}
   Rational add_rational(Rational r1, Rational r2) {
   Rational result;
  result.numerator = r1.numerator * r2.denominator + r2.numerator * r1.denominator;
  result.denominator = r1.denominator * r2.denominator;
  return simplify_rational(result);
}
             Rational subtract_rational(Rational r1, Rational r2) {
             Rational result;
             result.numerator = r1.numerator * r2.denominator - r2.numerator * r1.denominator;
             result.denominator = r1.denominator * r2.denominator;
             return simplify_rational(result);
```

```
    }
        Rational multiply_rational(Rational r1, Rational r2) {
        Rational result;
        result.numerator = r1.numerator * r2.numerator;
        result.denominator = r1.denominator * r2.denominator;
        return simplify_rational(result);
    }
        Rational divide_rational(Rational r1, Rational r2) {
        if (r2. numerator == 0) {
        fprintf(stderr, "Error: Division by zero.\n");
         exit (EXIT_FAILURE);
    }
        Rational result;
        result.numerator = r1.numerator * r2.denominator;
        result.denominator = r1.denominator * r2.numerator;
        return simplify_rational(result);
  }
   void print_rational(Rational r) {
   printf("%d/%d\n", r.numerator, r.denominator);
}
int main () {
        Rational r1 = {1, 2};
        Rational r2 = {2, 3};
        Rational sum = add_rational(r1, r2);
        print_rational(sum);
        Rational difference = subtract_rational(r1, r2);
        print_rational(difference);
        Rational product = multiply_rational(r1, r2);
        print_rational(product);
        Rational quotient = divide_rational(r1, r2);
        print_rational(quotient);
 return 0;
}
```

OUTPUT:-

```
Output
/tmp/zwBEXKA4Sm.o
7/6
-1/6
1/3
3/4


=== Code Execution Successful ===
```

Name: Date:

Class: MCA-II Roll No.:

Subject: Data Structure

Experiment No. 2

Aim: - Implement a complex number as a new data type.

Solution:

```
#include <stdio.h>
typedef struct {
    double real;
    double imaginary;
} Complex;
Complex create_complex(double realPart, double imaginaryPart) {
    return (Complex){realPart, imaginaryPart};
}
Complex add_complex(Complex c1, Complex c2) {
    return (Complex){c1.real + c2.real, c1.imaginary + c2.imaginary};
}
Complex subtract_complex(Complex c1, Complex c2) {
    return (Complex){c1.real - c2.real, c1.imaginary - c2.imaginary};
}
Complex multiply_complex(Complex c1, Complex c2) {
    return (Complex){c1.real * c2.real - c1.imaginary * c2.imaginary,
                     c1.real * c2.imaginary + c1.imaginary * c2.real};
}
void print_complex(Complex c) {
    printf("%.2f %c %.2fi\n", c.real, (c.imaginary >= 0) ? '+' : '-', (c.imaginary >= 0) ? '+' : '-', (c.imaginary >=
0) ? c.imaginary : -c.imaginary);
}
int main() {
    Complex c1 = create_complex(2, 3);
    Complex c2 = create_complex(1, -2);
    printf("Sum: ");
    print_complex(add_complex(c1, c2));
    printf("Difference: ");
    print_complex(subtract_complex(c1, c2));
    printf("Product: ");
    print_complex(multiply_complex(c1, c2));
    return 0;
}
```

OUTPUT: -

```
Output

/tmp/sDKNASmulR.o
Sum: 3.00 + 1.00i
Difference: 1.00 + 5.00i
Product: 8.00 - 1.00i
```

Name: **Date:**

Class: MCA-II **Roll No.:**

Subject: Data Structure

Experiment No. 3

Aim: - Write your own function for string operation.

Solution:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char* reverseString(const char* input) {
    int length = strlen(input);
    char* reversed = (char*)malloc((length + 1) * sizeof(char));
    if (reversed == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    for (int i = length - 1, j = 0; i >= 0; i--, j++) {
        reversed[j] = input[i];
    }
    reversed[length] = '\0';
    return reversed;
}
int main() {
    const char* input = "Hello, World!";
    char* reversed = reverseString(input);
    printf("Original string: %s\n", input);
    printf("Reversed string: %s\n", reversed);
    free(reversed);
    return 0;
}
```

OUTPUT: -

```
Output
/tmp/HpfEUTxDfl.o
Original string: Hello, World!
Reversed string: !dlroW ,olleH


=== Code Execution Successful ===
```

Name: **Date:**

Class: MCA-II **Roll No.:**

Subject: Data Structure

Experiment No. 4

Aim: - Implement a matrix as a new data type

Solution:

```
#include <stdio.h> int main()
 {
     int arr[3][4] = { { 1, 2, 3, 4 },
                       { 5, 6, 7, 8 },
                       { 9, 10, 11, 12 } };

     for (int i = 0; i < 3;
        i++) { for (int j =
        0; j < 4; j++) {
           printf("%d ", arr[i][j]);
        }
        printf("\n");
     }
     return 0;
 }
```

OUTPUT

```
/tmp/NWxUnssfeP.o
1 2 3 4
5 6 7 8
9 10 11 12


=== Code Execution Successful ===
```

	Name:	**Date:**
	Class: MCA-II	**Roll No.:**
	Subject: Data Structure	

Experiment No. 5

Aim: - Implement Stack using array.

Sol:-

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

int stack[10]; // Stack array
int n = 10;    // Size of the stack
int top = -1;  // Top of the stack

// Function to push an element onto the stack
void push(int val) {
    if (top >= n - 1) {
        printf("stack overflow\n");
    } else {
        top++;
        stack[top] = val;
    }
}
// Function to pop an element from the stack
void pop() {
    if (top <= -1) {
        printf("stack underflow\n");
    } else {
        printf("The popped element is %d\n", stack[top]);
        top--;
    }
}
// Function to display the elements in the stack
void display() {
    int i;
    if (top >= 0) {
        printf("stack elements are: ");
        for (i = top; i >= 0; i--) {
            printf("%d ", stack[i]);
        }
        printf("\n");
    } else {
        printf("stack is empty\n");
    }
}
```

```
int main() {
    int ch, val;

    // Display menu options
    printf("1) Push in stack\n");
    printf("2) Pop from stack\n");
    printf("3) Display stack\n");
    printf("4) Exit stack\n");

    do {
        printf("Enter your choice: ");
        scanf("%d", &ch);

        switch (ch) {
            case 1:
                printf("Enter value to be pushed: ");
                scanf("%d", &val);
                push(val);
                break;
            case 2:
                pop();
                break;
            case 3:
                display();
                break;
            case 4:
                printf("Exiting stack\n");
                break;
            default:
                printf("Invalid choice\n");
        }
    } while (ch != 4);

    return 0;
}
```

OUTPUT:-

```
/tmp/dfV1Yh6ymS.o
1) Push in stack
2) Pop from stack
3) Display stack
4) Exit stack
Enter your choice: 1
Enter value to be pushed: 2
Enter your choice: 3
stack elements are: 2
Enter your choice: 
```

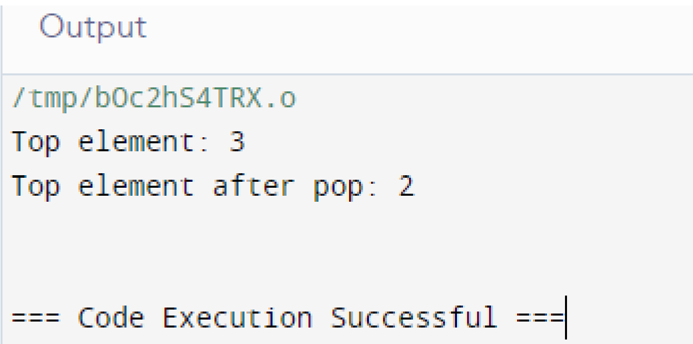
Experiment No. 6

Aim: Implementation of a stack using a linked list.
Solution:

```
#include <stdio.h>
#include <stdlib.h>
// Node structure
                typedef struct Node {
                  int data;
                  struct Node* next;
        } Node;
// Stack structure
                typedef struct {
                Node* topNode;
        } Stack;
// Function to create a new node
                Node* createNode(int val) {
                    Node* newNode = (Node*)malloc(sizeof(Node));
                    if (newNode == NULL) {
                    printf("Memory allocation failed.\n");
                    exit(EXIT_FAILURE);
    }
                newNode->data = val;
                newNode->next = NULL;
                return newNode;
    }
// Function to initialize a stack
        void initializeStack(Stack* stack) {
            stack->topNode = NULL;
      }
// Function to push an element onto the stack
          void push(Stack* stack, int val) {
              Node* newNode = createNode(val);
              newNode->next = stack->topNode;
              stack->topNode = newNode;
      }
// Function to pop an element from the stack
          void pop(Stack* stack) {
             if (stack->topNode == NULL) {
             printf("Stack underflow!\n");
             return;
      }
          Node* temp = stack->topNode;
```

```
        stack->topNode = stack->topNode->next;
        free(temp);
}
// Function to get the top element of the stack
          int top(const Stack* stack) {
             if (stack->topNode == NULL) {
     printf("Stack is empty!\n");
     return 0;
  }
  return stack->topNode->data;
}
// Function to check if the stack is empty
          int isEmpty(const Stack* stack) {
          return stack->topNode == NULL;
}
int main() {
  // Declare and initialize a stack
          Stack stack;
          initializeStack(&stack);
  // Push elements onto the stack
          push(&stack, 1);
          push(&stack, 2);
          push(&stack, 3);
  // Print the top element of the stack
          printf("Top element: %d\n", top(&stack));
  // Pop an element from the stack
          pop(&stack);
          printf("Top element after pop: %d\n", top(&stack));
  // Pop more elements than pushed to test underflow
          pop(&stack);
          pop(&stack);
          return 0;
 }
```

OUTPUT: -



```
Output
/tmp/bOc2hS4TRX.o
Top element: 3
Top element after pop: 2


=== Code Execution Successful ===
```

Name: Date:

Class: MCA-II Roll No.:

Subject: Data Structure

Experiment No. 7

Aim: Use of stack for checking brackets in mathematical expression.
Solution:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
    // Function to check whether two characters are opening and closing of the same type.
int ArePair(char opening, char closing) {
   if (opening == '(' && closing == ')')
      return 1;
   else if (opening == '{' && closing == '}')
      return 1;
   else if (opening == '[' && closing == ']')
      return 1;
   return 0;
}
// Function to check whether parentheses are balanced in the expression.
int AreParenthesesBalanced(char *exp) {
   int i;
   int len = strlen(exp);
   char *S = (char *)malloc(len * sizeof(char));
   int top = -1;

   for (i = 0; i < len; i++) {
      if (exp[i] == '(' || exp[i] == '{' || exp[i] == '[') {
         S[++top] = exp[i];
      } else if (exp[i] == ')' || exp[i] == '}' || exp[i] == ']') {
         if (top == -1 || !ArePair(S[top], exp[i])) {
            free(S);
            return 0;
         } else {
            top--;
         }
      }
   }
   free(S);
   return top == -1;
}
int main() {
   char expression[100];
   printf("Enter an expression: ");
   scanf("%s", expression);
```

```
    if (AreParenthesesBalanced(expression)) {
        printf("Balanced\n");
    } else {
        printf("Not Balanced\n");
    }

    return 0;
}
```

OUTPUT: -

```
/tmp/JLF80pJvnT.o
Enter an expression: 25
Balanced


=== Code Execution Successful ===
```

Experiment No. 9

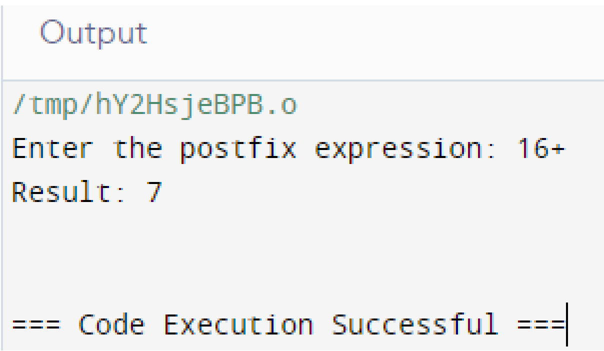
Aim: Evaluate a postfix expression.

Solution:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define STACK_SIZE 100
// Structure to represent a stack
                typedef struct {
                    int items[STACK_SIZE];
                    int top;
} Stack;
// Initialize stack
        void initializeStack(Stack* stack) {
        stack->top = -1;
}
// Check if the stack is empty
       int isEmpty(Stack* stack) {
       return stack->top == -1;
}
// Check if the stack is full
      int isFull(Stack* stack) {
       return stack->top == STACK_SIZE - 1;
}
// Push an item onto the stack
     void push(Stack* stack, int item) {
      if (isFull(stack)) {
        printf("Stack overflow\n");
        exit(EXIT_FAILURE);
    }
    stack->items[++stack->top] = item;
}
// Pop an item from the stack
                int pop(Stack* stack) {
                if (isEmpty(stack)) {
                printf("Stack underflow\n");
                exit(EXIT_FAILURE);
    }
                return stack->items[stack->top--];
}
// Function to evaluate a postfix expression
                int evaluatePostfix(const char* postfix) {
                Stack operands;
                initializeStack(&operands);
                int i = 0;
    while (postfix[i] != '\0') {
        char ch = postfix[i++];
        if (isdigit(ch)) {
```

```
            push(&operands, ch - '0'); // Convert char to int
        } else if (ch == '+' || ch == '-' || ch == '*' || ch == '/') {
            int operand2 = pop(&operands);
            int operand1 = pop(&operands);
            switch (ch) {
                case '+':
                    push(&operands, operand1 + operand2);
                    break;
                case '-':
                    push(&operands, operand1 - operand2);
                    break;
                case '*':
                    push(&operands, operand1 * operand2);
                    break;
                case '/':
                    push(&operands, operand1 / operand2);
                    break;
            }
        }
    }
    return pop(&operands);
}
int main() {
    char postfix_expression[100];
    printf("Enter the postfix expression: ");
    fgets(postfix_expression, sizeof(postfix_expression), stdin);
    postfix_expression[strcspn(postfix_expression, "\n")] = '\0'; // Remove newline character
    int result = evaluatePostfix(postfix_expression);
    printf("Result: %d\n", result);
    return 0;
}
```

OUTPUT: -



Name: **Date:**

Class: MCA-II **Roll No.:**

Subject: Data Structure

Experiment No. 8

Aim: - Conversion of an Infix to a postfix expression.

Sol: -

```
#include <stdio.h>
#include <ctype.h>
char stack[20];
int top = -1;

void push(char x) { stack[++top] = x; }
char pop() { return (top == -1) ? -1 : stack[top--]; }
int priority(char x) { return (x == '(') ? 0 : ((x == '+' || x == '-') ? 1 : ((x == '*' || x == '/') ? 2 : 0)); }

int main() {
   char exp[20], *e, x;
   printf("Enter the infix expression: ");
   scanf("%s", exp);
   e = exp;
   printf("Postfix expression is: ");
   while (*e) {
      if (isalnum(*e))
         printf("%c", *e);
      else if (*e == '(')
         push(*e);
      else if (*e == ')') {
         while ((x = pop()) != '(')
            printf("%c", x);
      } else {
         while (priority(stack[top]) >= priority(*e))
            printf("%c", pop());
         push(*e);
      }
      e++;
   }
   while (top != -1)
      printf("%c", pop());
   return 0;
}
```

Output: -

```
/tmp/HvGRc3VgLz.o
Enter the infix expression: 12
Postfix expression is: 12

=== Code Execution Successful ===
```

Experiment No. 10

Aim: - Implementation of queue using array.

Solution:-

```
#include <stdio.h>
#include <stdlib.h>
int queue[100];
int n = 100;
int front = -1;
int rear = -1;
void Insert() {
   int val;
   if (rear == n - 1) {
      printf("Queue Overflow\n");
   } else {
      if (front == -1) {
         front = 0;
      }
      printf("Insert the element in queue: ");
      scanf("%d", &val);
      rear++;
      queue[rear] = val;
   }
}
void Delete() {
   if (front == -1 || front > rear) {
      printf("Queue Underflow\n");
      return;
   } else {
      printf("Element deleted from queue is: %d\n", queue[front]);
      front++;
   }
}
void Display() {
   int i;
   if (front == -1) {
      printf("Queue is empty\n");
   } else {
      printf("Queue elements are: ");
      for (i = front; i <= rear; i++) {
         printf("%d ", queue[i]);
      }
      printf("\n");
   }
}
int main() {
   int ch;
   printf("1) Insert element to queue\n");
   printf("2) Delete element from queue\n");
   printf("3) Display all the elements of queue\n");
```

```
    printf("4) Exit\n");
    do {
        printf("Enter your choice: ");
        scanf("%d", &ch);
        switch (ch) {
            case 1:
                Insert();
                break;
            case 2:
                Delete();
                break;
            case 3:
                Display();
                break;
            case 4:
                printf("Exit\n");
                break;
            default:
                printf("Invalid choice\n");
                break;
        }
    } while (ch != 4);
    return 0;
}
```

OUTPUT

```
/tmp/EWcZkZbZ4S.o
1) Insert element to queue
2) Delete element from queue
3) Display all the elements of queue
4) Exit
Enter your choice: 1
Insert the element in queue: 3
Enter your choice: 2
Element deleted from queue is: 3
Enter your choice:
```

Experiment No. 11

Aim: - Implementation of queue using linked list.

Sol:-

```
#include <stdio.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
};
struct node *front = NULL;
struct node *rear = NULL;
struct node *temp;
void Insert() {
   int val;
   printf("Insert the element in queue : \n");
   scanf("%d", &val);
   if (rear == NULL) {
      rear = (struct node *)malloc(sizeof(struct node));
      rear->next = NULL;
      rear->data = val;
      front = rear;
   } else {
      temp = (struct node *)malloc(sizeof(struct node));
      rear->next = temp;
      temp->data = val;
      temp->next = NULL;
      rear = temp;
   }
}
void Delete() {
   temp = front;
   if (front == NULL) {
      printf("Underflow\n");
      return;
   } else if (temp->next != NULL) {
      temp = temp->next;
      printf("Element deleted from queue is : %d\n", front->data);
      free(front);
      front = temp;
   } else {
      printf("Element deleted from queue is : %d\n", front->data);
      free(front);
      front = NULL;
      rear = NULL;
   }  }
void Display() {
   temp = front;
   if (front == NULL && rear == NULL) {
      printf("Queue is empty\n");
```

```
        return;
    }
    printf("Queue elements are: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
int main() {
    int ch;
    printf("1) Insert element to queue\n");
    printf("2) Delete element from queue\n");
    printf("3) Display all the elements of queue\n");
    printf("4) Exit\n");
    do {
        printf("Enter your choice : \n");
        scanf("%d", &ch);
        switch (ch) {
            case 1:
                Insert();           break;
            case 2:
                Delete();           break;
            case 3:
                Display();          break;
            case 4:
                printf("Exit\n");           break;
            default:
                printf("Invalid choice\n");
        }
    } while (ch != 4);
    return 0;
}
```

OUTPUT: -

```
/tmp/7PzhBr9bZw.o
1) Insert element to queue
2) Delete element from queue
3) Display all the elements of queue
4) Exit
Enter your choice :
1
Insert the element in queue :
1
Enter your choice :
3
Queue elements are: 1
Enter your choice :
```

Experiment No. 12

Aim: - Implementation of Priority queue.

Sol:-

```
#include <stdio.h>
#include <stdlib.h>
typedef struct node {
   int priority;
   int info;
   struct node *link;
} node;
typedef struct {
   node *front;
} Priority_Queue;

void init_pq(Priority_Queue *pq) {
   pq->front = NULL;
}
void insert(Priority_Queue *pq, int item, int priority) {
   node *tmp, *q;
   tmp = (node *)malloc(sizeof(node));
   tmp->info = item;
   tmp->priority = priority;

   if (pq->front == NULL || priority < pq->front->priority) {
      tmp->link = pq->front;
      pq->front = tmp;
   } else {
      q = pq->front;
      while (q->link != NULL && q->link->priority <= priority)
         q = q->link;
      tmp->link = q->link;
      q->link = tmp;
   }
}
void del(Priority_Queue *pq) {
   node *tmp;
   if (pq->front == NULL)
      printf("Queue Underflow\n");
   else {
      tmp = pq->front;
      printf("Deleted item is: %d\n", tmp->info);
      pq->front = pq->front->link;
      free(tmp);
   }
}
void display(Priority_Queue *pq) {
   node *ptr;
   ptr = pq->front;
   if (pq->front == NULL)
```

```
        printf("Queue is empty\n");
    else {
        printf("Queue is :\n");
        printf("Priority\tItem\n");
        while (ptr != NULL) {
            printf("%d\t\t%d\n", ptr->priority, ptr->info);
            ptr = ptr->link;
        }
    }
}
int main() {
    int choice, item, priority;
    Priority_Queue pq;
    init_pq(&pq);
    do {
        printf("1.Insert\n");
        printf("2.Delete\n");
        printf("3.Display\n");
        printf("4.Quit\n");
        printf("Enter your choice : ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Input the item value to be added in the queue : ");
                scanf("%d", &item);
                printf("Enter its priority : ");
                scanf("%d", &priority);
                insert(&pq, item, priority);
                break;
            case 2:
                del(&pq);
                break;
            case 3:
                display(&pq);
                break;
            case 4:
                break;
            default:
                printf("Wrong choice\n");
        }
    } while (choice != 4);
    return 0;
}
```

Output:

```
/tmp/6zJCRxGAkQ.o
1.Insert
2.Delete
3.Display
4.Quit
Enter your choice : 1
Input the item value to be added in the queue : 10
Enter its priority : 3
```

Experiment No. 13

Aim: - Implementation of Deque.

Sol:-

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10
struct dequeue {
   int a[20];
   int f, r;
};
void init_dequeue(struct dequeue *dq) {
   dq->f = -1;
   dq->r = -1;
}
void insert_at_end(struct dequeue *dq, int item) {
   if (dq->r >= SIZE - 1) {
      printf("\nInsertion is not possible, overflow!!!!");
   } else {
      if (dq->f == -1) {
         dq->f++;
         dq->r++;
      } else {
         dq->r = dq->r + 1;
      }
      dq->a[dq->r] = item;
      printf("\nInserted item is %d", dq->a[dq->r]);
   }
}
void insert_at_beg(struct dequeue *dq, int item) {
   if (dq->f == -1) {
      dq->f = 0;
      dq->a[++dq->r] = item;
      printf("\nInserted element is: %d", item);
   } else if (dq->f != 0) {
      dq->a[--dq->f] = item;
      printf("\nInserted element is: %d", item);
   } else {
      printf("\nInsertion is not possible, overflow!!!");
   }
}
void delete_fr_front(struct dequeue *dq) {
   if (dq->f == -1) {
      printf("Deletion is not possible::dequeue is empty");
      return;
   } else {
      printf("The deleted element is: %d", dq->a[dq->f]);
      if (dq->f == dq->r) {
         dq->f = dq->r = -1;
```

```
            return;
        } else {
            dq->f = dq->f + 1;
        }
    }
}
void delete_fr_rear(struct dequeue *dq) {
    if (dq->f == -1) {
        printf("Deletion is not possible::dequeue is empty");
        return;
    } else {
        printf("The deleted element is: %d", dq->a[dq->r]);
        if (dq->f == dq->r) {
            dq->f = dq->r = -1;
        } else {
            dq->r = dq->r - 1;
        }
    }
}
void show(struct dequeue *dq) {
    int i;
    if (dq->f == -1) {
        printf("Dequeue is empty");
    } else {
        for (i = dq->f; i <= dq->r; i++) {
            printf("%d ", dq->a[i]);
        }
    }
}
int main() {
    int c, i;
    struct dequeue d;
    init_dequeue(&d);
    do {
        printf("\n1. Insert at beginning");
        printf("\n2. Insert at end");
        printf("\n3. Show");
        printf("\n4. Deletion from front");
        printf("\n5. Deletion from rear");
        printf("\n6. Exit");
        printf("\nEnter your choice:");
        scanf("%d", &c);
        switch (c) {
            case 1:
                printf("Enter the element to be inserted: ");
                scanf("%d", &i);
                insert_at_beg(&d, i);
                break;
            case 2:
                printf("Enter the element to be inserted: ");
                scanf("%d", &i);
                insert_at_end(&d, i);
                break;
            case 3:
                show(&d);
```

```
            break;
        case 4:
            delete_fr_front(&d);
            break;
        case 5:
            delete_fr_rear(&d);
            break;
        case 6:
            exit(0);
            break;
        default:
            printf("Invalid choice");
            break;
        }
    } while (c != 6);
    return 0;
}
```

OUTPUT:-

```
/tmp/i3sSCsEIML.o

1. Insert at beginning
2. Insert at end
3. Show
4. Deletion from front
5. Deletion from rear
6. Exit
Enter your choice:2
Enter the element to be inserted: 12
```

Name: **Date:**

Class: MCA-II **Roll No.:**

Subject: Data Structure

Experiment No. 14

Aim: - Implementation of circular queue.

Sol:-

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 5
int cqueue[MAX_SIZE];
int front = -1, rear = -1;
void insertCQ(int val) {
   if ((front == 0 && rear == MAX_SIZE - 1) || (front == rear + 1)) {
      printf("Queue Overflow\n");
      return;
   }
   if (front == -1) {
      front = 0;
      rear = 0;
   } else {
      if (rear == MAX_SIZE - 1)
         rear = 0;
      else
         rear = rear + 1;
   }
   cqueue[rear] = val;
}
void deleteCQ() {
   if (front == -1) {
      printf("Queue Underflow\n");
      return;
   }
   printf("Element deleted from queue is : %d\n", cqueue[front]);
   if (front == rear) {
      front = -1;
      rear = -1;
   } else {
      if (front == MAX_SIZE - 1)
         front = 0;
      else
         front = front + 1;
   }}
void displayCQ() {
   int f = front, r = rear;
   if (front == -1) {
      printf("Queue is empty\n");
      return;
   }
   printf("Queue elements are :\n");
   if (f <= r) {
      while (f <= r) {
         printf("%d ", cqueue[f]);
         f++;
```

```
        }
    } else {
        while (f <= MAX_SIZE - 1) {
            printf("%d ", cqueue[f]);
            f++;        }
        f = 0;
        while (f <= r) {
            printf("%d ", cqueue[f]);
            f++;
        }    }
    printf("\n");
}
int main() {
    int ch, val;
    printf("1) Insert\n");
    printf("2) Delete\n");
    printf("3) Display\n");
    printf("4) Exit\n");

    do {
        printf("Enter choice: ");
        scanf("%d", &ch);
        switch (ch) {
            case 1:
                printf("Input for insertion: ");
                scanf("%d", &val);
                insertCQ(val);
                break;
            case 2:
                deleteCQ();
                break;
            case 3:
                displayCQ();
                break;
            case 4:
                printf("Exit\n");
                break;
            default:
                printf("Incorrect choice!\n");
                getchar(); // to capture the extra newline character
                break;}
    } while (ch != 4);
    return 0;
}
```

OUTPUT:-

```
/tmp/ffubC31czV.o
1) Insert
2) Delete
3) Display
4) Exit
Enter choice: 1
Input for insertion: 25
Enter choice: 3
Queue elements are :
25
Enter choice: 
```

Name: **Date:**

Class: MCA-II **Roll No.:**

Subject: Data Structure

Experiment No. 15

Aim: Implementation of singly linked list.

Sol:-

```
#include <stdio.h>
#include <stdlib.h>
struct node {
   int data;
   struct node* next;
};
struct node* head = NULL;
void insert(int new_data) {
   struct node *new_node = (struct node*) malloc (sizeof(struct node));
   new_node->data = new_data;
   new_node->next = head;
   head = new_node;
}
void display() {
   struct node *ptr;
   ptr = head;
   while (ptr != NULL) {
      printf("%d ", ptr->data);
      ptr = ptr->next;
   }
}
int main() {
   insert(3);
   insert(1);
   insert(7);
   insert(2);
   insert(9);
   printf("The linked list is: ");
   display();
   return 0;
}
```

OUTPUT:-

```
/tmp/rM4a40bpwU.o
The linked list is: 9 2 7 1 3

=== Code Execution Successful ===
```

Name: Date:

Class: MCA-II Roll No.:

Subject: Data Structure

Experiment No. 16

Aim: Implementation of singly circularly linked list.

Sol:-

```
#include <stdio.h>
#include <stdlib.h>
struct node {
   int data;
   struct node* next;
};
struct node* head = NULL;
void insert(int new_data) {
   struct node *new_node = (struct node*) malloc(sizeof(struct node));
   new_node->data = new_data;
   new_node->next = head ? head : new_node;
   if (head) {
      struct node *ptr = head;
      while (ptr->next != head)
         ptr = ptr->next;
      ptr->next = new_node;
   }
   head = new_node;
}
void display() {
   struct node *ptr = head;
   if (head) {
      do {
         printf("%d ", ptr->data);
         ptr = ptr->next;
      } while (ptr != head);
   }}
int main() {
   insert(4);    insert(6);
   insert(8);   insert(9);
   insert(6);
   printf("The circular singly linked list is: ");
   display();
   return 0;
}
```

OUTPUT:

```
/tmp/JAajixhv6K.o
The circular singly linked list is: 6 9 8 6 4

=== Code Execution Successful ===
```

Name: Date:

Class: MCA-II Roll No.:

Subject: Data Structure

Experiment No. 17

Aim: Implementation of doubly circularly linked list.

Sol:-

```
#include <stdio.h>
#include <stdlib.h>
struct node {
    int data;
    struct node* prev;
    struct node* next;
};
struct node* head = NULL;
void insert(int new_data) {
    struct node* new_node = (struct node*) malloc(sizeof(struct node));
    new_node->data = new_data;
    new_node->prev = NULL;
    new_node->next = head;
    if (head != NULL) {
        head->prev = new_node;
    }
    head = new_node;
}
void display() {
    struct node* ptr;
    ptr = head;
    while (ptr != NULL) {
        printf("%d ", ptr->data);
        ptr = ptr->next;
    }
}
int main() {
    insert(2);    insert(4);
    insert(5);    insert(9);
    insert(8);
    printf("The doubly linked list is: ");
    display();
    return 0;
}
```

OUTPUT:-

```
/tmp/PBTO0k0tQo.o
The doubly linked list is: 8 9 5 4 2

=== Code Execution Successful ===
```

Experiment No. 22

Aim: Implementation of hash Collision resolution techniques.

Sol:-

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

#define T_S 5

typedef enum { Legi, Emp } EntryType;

typedef struct {
   int e;
   EntryType info;
} Entry;

typedef struct {
   int size;
   Entry *table;
} HashTable;

int Hash(int k, int size) { return k % size; }

HashTable *Initiate(int size) {
   if (size < T_S) {
      printf("Table Size is Too Small\n");
      return NULL;
   }
   HashTable *ht = (HashTable *)malloc(sizeof(HashTable));
   if (!ht) {
      printf("Out of Space\n");
      return NULL;
   }
   ht->size = size;
   ht->table = (Entry *)calloc(size, sizeof(Entry));
   if (!ht->table) {
      printf("Table Size is Too Small\n");
      return NULL;
   }
   return ht;
}
```

```
int FindKey(int k, HashTable *ht) {
    int hash = Hash(k, ht->size);
    while (ht->table[hash].info != Emp && ht->table[hash].e != k) {
        hash = (hash + 1) % ht->size;
    }
    return hash;
}

void Insert(int k, HashTable *ht) {
    int pos = FindKey(k, ht);
    if (ht->table[pos].info != Legi) {
        ht->table[pos].info = Legi;
        ht->table[pos].e = k;
    }
}

void Display(HashTable *ht) {
    for (int i = 0; i < ht->size; i++) {
        int v = ht->table[i].e;
        printf("Position: %d Element: %s\n", i + 1, (v == 0) ? "Null" : (v == -1 ? "Deleted" : "Legitimate"));
    }
}

HashTable *Rehash(HashTable *ht) {
    Entry *oldTable = ht->table;
    int oldSize = ht->size;
    ht = Initiate(2 * oldSize);
    for (int i = 0; i < oldSize; i++) {
        if (oldTable[i].info == Legi)
            Insert(oldTable[i].e, ht);
    }
    free(oldTable);
    return ht;
}

int main() {
    int value, size, i = 1, choice;
    HashTable *ht;
    while (1) {
        printf("\n1. Initialize size of the table\n");
        printf("2. Insert element into the table\n");
        printf("3. Display Hash Table\n");
        printf("4. Rehash Hash Table\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
```

```
        switch (choice) {
            case 1:
                printf("Enter size of the Hash Table: ");
                scanf("%d", &size);
                ht = Initiate(size);
                break;
            case 2:
                if (i > ht->size) {
                    printf("Table is Full, Rehash the table\n");
                    continue;
                }
                printf("Enter element to be inserted: ");
                scanf("%d", &value);
                Insert(value, ht);
                i++;
                break;
            case 3:
                Display(ht);
                break;
            case 4:
                ht = Rehash(ht);
                break;
            case 5:
                exit(0);
            default:
                printf("\nEnter correct option\n");
        }
    }
    return 0;
}
```

OUTPUT:-

```
1. Initialize size of the table
2. Insert element into the table
3. Display Hash Table
4. Rehash Hash Table
5. Exit
Enter your choice: 1
Enter size of the Hash Table: 5

1. Initialize size of the table
2. Insert element into the table
3. Display Hash Table
4. Rehash Hash Table
5. Exit
Enter your choice: 3
Position: 1 Element: Null
Position: 2 Element: Null
Position: 3 Element: Null
Position: 4 Element: Null
Position: 5 Element: Null
```

Experiment No. 18

Aim: Solving Poly nominal arithmetic using linked list.

Sol: -

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
   int coef;
   int exp;
   struct Node* next;
} Node;
void insert(Node** poly, int coef, int exp) {
   Node* temp = (Node*)malloc(sizeof(Node));
   temp->coef = coef;
   temp->exp = exp;
   temp->next = *poly;
   *poly = temp;
}
void print(Node* poly) {
   while (poly != NULL) {
      printf("%dx^%d", poly->coef, poly->exp);
      if (poly->next != NULL) printf(" + ");
      poly = poly->next;
   }
   printf("\n");
}
Node* add(Node* poly1, Node* poly2) {
   Node* result = NULL;
   while (poly1 != NULL || poly2 != NULL) {
      if (poly1 && (!poly2 || poly1->exp > poly2->exp)) {
         insert(&result, poly1->coef, poly1->exp);
         poly1 = poly1->next;
      } else if (poly2 && (!poly1 || poly1->exp < poly2->exp)) {
         insert(&result, poly2->coef, poly2->exp);
         poly2 = poly2->next;
      } else {
         insert(&result, poly1->coef + poly2->coef, poly1->exp);
         poly1 = poly1->next;
         poly2 = poly2->next;
      }
   }
   return result;
```

```
}
int main() {
    Node* poly1 = NULL;
    insert(&poly1, 5, 4);
    insert(&poly1, 3, 2);
    insert(&poly1, 1, 0);
    Node* poly2 = NULL;
    insert(&poly2, 4, 4);
    insert(&poly2, 2, 2);
    insert(&poly2, 1, 1);
    printf("First polynomial: ");
    print(poly1);
    printf("Second polynomial: ");
    print(poly2);
    Node* result = add(poly1, poly2);
    printf("Result: ");
    print(result);
    return 0;
}
```

OUTPUT:

```
/tmp/5rYuBv65Ik.o
First polynomial: 1x^0 + 3x^2 + 5x^4
Second polynomial: 1x^1 + 2x^2 + 4x^4
Result: 5x^4 + 3x^2 + 1x^0 + 4x^4 + 2x^2 + 1x^1
```

YSPM	**Name:**	**Date:**
	Class: MCA-II	**Roll No.:**
	Subject: Data Structure	

Experiment No. 19

Aim: Implementation of binary tree and operation.

Sol: -

```
#include<stdio.h>
#include<stdlib.h>
typedef struct node {
   int value;
   struct node *left, *right;
} Node;
Node* newNode(int value) {
   Node* temp = (Node*)malloc(sizeof(Node));
   temp->value = value;
   temp->left = temp->right = NULL;
   return temp;
}
void inorder(Node* root) {
   if (root) {
      inorder(root->left);
      printf("%d ", root->value); // Print without newline
      inorder(root->right);
   } }
Node* insert(Node* root, int value) {
   if (!root) return newNode(value);
   if (value < root->value) root->left = insert(root->left, value);
   else if (value > root->value) root->right = insert(root->right, value);
   return root;
}
int main() {
   Node* root = NULL;
   int values[] = {10, 10, 30, 25, 36, 56, 78};
   printf("TechVidvan Tutorial: Implementation of a Binary Tree in C!\n\n");
   for (int i = 0; i < sizeof(values) / sizeof(values[0]); i++)
      root = insert(root, values[i]);    inorder(root);
   return 0;
}
```

OUTPUT

```
/tmp/98r9tdKO8q.o
TechVidvan Tutorial: Implementation of a Binary Tree in C!

10 25 30 36 56 78
```

Name: **Date:**

Class: MCA-II **Roll No.:**

Subject: Data Structure

Experiment No. 20

Aim: Traversal of binary tree.

Sol: -

```
#include <stdio.h>
#include <stdlib.h>
struct node {
   int data;
   struct node* left;
   struct node* right;
};
struct node* newNode(int data) {
   struct node* node = (struct node*)malloc(sizeof(struct node));
   node->data = data;
   node->left = node->right = NULL;
   return node;
}
void printInorder(struct node* node) {
   if (node) {
      printInorder(node->left);
      printf("%d ", node->data);
      printInorder(node->right);
   }
}
int main() {
   struct node* root = newNode(1);
   root->left = newNode(2);
   root->right = newNode(3);
   root->left->left = newNode(4);
   root->left->right = newNode(5);
   printf("Inorder traversal of binary tree is \n");
   printInorder(root);
   return 0;
}
```

OUTPUT

```
/tmp/cuDTUP4R3U.o
Inorder traversal of binary tree is
4 2 5 1 3

=== Code Execution Successful ===
```

Experiment No. 21

Aim: Implementation of hash table.

Sol: -

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
    int data;
    struct Node* next;
} Node;
typedef struct HashTable {
    int total_elements;
    Node** table;
} HashTable;
int getHash(int key, int total_elements) {
    return key % total_elements;
}
HashTable* createHashTable(int n) {
    HashTable* ht = (HashTable*)malloc(sizeof(HashTable));
    ht->total_elements = n;
    ht->table = (Node**)calloc(n, sizeof(Node*));
    return ht;
}
void insertElement(HashTable* ht, int key) {
    int index = getHash(key, ht->total_elements);
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = key;
    newNode->next = ht->table[index];
    ht->table[index] = newNode;
}
void removeElement(HashTable* ht, int key) {
    int index = getHash(key, ht->total_elements);
    Node* current = ht->table[index];
    Node* prev = NULL;
    while (current) {
        if (current->data == key) {
            if (prev)
                prev->next = current->next;
            else
                ht->table[index] = current->next;
            free(current);
            return;
        }
```

```
        prev = current;
        current = current->next;
    }}
void printAll(HashTable* ht) {
    for (int i = 0; i < ht->total_elements; i++) {
        printf("Index %d: ", i);
        Node* current = ht->table[i];
        while (current) {
            printf("%d => ", current->data);
            current = current->next;
        }
        printf("\n");
    }  }
int main() {
    HashTable* ht = createHashTable(3);
    int arr[] = {2, 4, 6, 8, 10};
    for (int i = 0; i < 5; i++)
        insertElement(ht, arr[i]);
    printf("..:: Hash Table ::..\n");
    printAll(ht);
    removeElement(ht, 4);
    printf("\n..:: After deleting 4 ::..\n");
    printAll(ht);
    return 0;
}
```

OUTPUT

```
..:: Hash Table ::..
Index 0: 6 =>
Index 1: 10 => 4 =>
Index 2: 8 => 2 =>

..:: After deleting 4 ::..
Index 0: 6 =>
Index 1: 10 =>
Index 2: 8 => 2 =>


=== Code Execution Successful ===
```