# DATA STRUCTURES LAB MANUAL SOLUTIONS

**SOUNDARYALAKSH**
**MI S**

**230701326**

**CSE-E BATCH:2023-**

**2024 Experiment 1** :

IMPLEMENTATION

OF SINGLY

LINKED LIST

1. Start
2. Create a structure and functions for each operations
3. Display the main menu
4. Read user choice
5. Execute choice operation
6. Display operation completion
7. Back to main menu
8. Check for exit, if no Execute the operation for the given choice
9. Otherwise end Program:

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

// Define the structure for a node in the linked list
struct Node {     int data;     struct Node* next;
};

// Function to create a new node struct Node* createNode(int value) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
if (newNode == NULL) {
       printf("Memory allocation failed\n");
exit(1);
   }
   newNode->data = value;     newNode-
>next = NULL;
return newNode;
}
```

```
// Function to insert a node at the beginning of the list void
insertAtBeginning(struct Node** head, int value) {     struct Node* newNode =
createNode(value);     newNode->next
= *head;
   *head = newNode;
}

// Function to insert a node after a given node void insertAfter(struct
Node* prevNode, int value) {     if
(prevNode == NULL) {
      printf("Previous node cannot be NULL\n");
return;
   }
  struct Node* newNode = createNode(value);     newNode-
>next = prevNode->next;
   prevNode->next = newNode;
}

// Function to insert a node at the end of the list void
insertAtEnd(struct Node** head, int value) {     struct Node*
newNode = createNode(value);
   if (*head == NULL) {
*head = newNode;
      return;
   }
   struct Node* temp = *head;
while (temp->next != NULL) {
      temp = temp->next;
   }
   temp->next = newNode;
}

// Function to find an element in the list struct Node*
findElement(struct Node* head, int value) {     struct
Node* current = head;     while (current != NULL) {        if
(current->data == value) {
         return current;
      }
      current = current->next;
   }
   return NULL;
}

// Function to find the next node after a given node
struct Node* findNext(struct Node* node) {     if (node
!= NULL && node->next != NULL) {
      return node->next;
```

```
    }
    return NULL;
}

// Function to find the previous node before a given node struct
Node* findPrevious(struct Node* head, struct Node* node) {     if
(head == node) {
return NULL;
    }
    struct Node* current = head;
    while (current != NULL && current->next != node) {
        current = current->next;
    }
    return current;
}

// Function to check if a node is the last node in the list int
isLast(struct Node* node) {
    return (node != NULL && node->next == NULL);
}

// Function to check if the list is empty int
isEmpty(struct Node* head) {
    return (head == NULL);
}

// Function to delete the first node in the list void
deleteFirst(struct Node** head) {     if (*head ==
NULL) {         printf("List is
empty\n");
        return;
    }
    struct Node* temp = *head;
*head = (*head)->next;     free(temp);
}

// Function to delete the node after a given node void
deleteAfter(struct Node* prevNode) {     if (prevNode
== NULL || prevNode->next == NULL) {
printf("Invalid previous node\n");
        return;
    }
    struct Node* temp = prevNode->next;     prevNode-
>next = temp->next;
free(temp);
}
```

```
// Function to delete the last node in the list void
deleteLast(struct Node** head) {     if (*head ==
NULL) {         printf("List is
empty\n");
        return;
    }
    if ((*head)->next == NULL) {
        free(*head);
*head = NULL;
return;
    }
    struct Node* secondLast = *head;     while (secondLast->next->next
!= NULL) {
        secondLast = secondLast->next;
    }
    free(secondLast->next);     secondLast>next
= NULL; }

// Function to delete the entire list
void deleteList(struct Node** head) {
struct Node* current = *head;     struct
Node* next;     while (current
!= NULL) {         next = current>next;
free(current);
        current = next;
    }
    *head = NULL;
}

// Function to display the list void
displayList(struct Node* head) {
struct Node* current = head;     while
(current != NULL) {         printf("%d -
> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}
int main() {
    struct Node* head = NULL;

    //          Insert          nodes
insertAtBeginning(&head, 10);
insertAtEnd(&head, 20);
    insertAfter(head, 10, 15);
```

```
    // Display the list
printf("Initial list:\n");
    displayList(head);

    // Find and display an element
int searchValue = 15;
    struct Node* searchResult = findElement(head, searchValue);
if (searchResult != NULL) {
        printf("Element %d found in the list\n", searchValue);
    } else {
        printf("Element %d not found in the list\n", searchValue);
    }

   // Delete nodes    deleteFirst(&head);
deleteAfter(head);
   deleteLast(&head);

    // Display the modified list
printf("List after deletions:\n");
displayList(head);

    // Delete the entire list
    deleteList(&head);

    return 0;
}
```

**Experiment 2:**

**IMPLEMENTATION**

**OF DOUBLY**

**LINKED LIST**

Algorithm:

1. Start
2. Create a structure and functions for each operations
3. Declare the variables
4. Create a do-while loop to display the menu and execute operations based on your input until the user chooses to exit
5. Inside the loop display the menu options
6. Prompt the user to enter their choice
7. Use switch statement to perform different operations based on the user’s choice and display it.
8. Repeat the loop until the user chooses to exit
9. Exit

Program:

```
#include <stdio.h>

#include <stdlib.h>

// Define a node structure for doubly linked list struct
Node {     int data;     struct Node* prev;     struct
Node* next;
};

// Function to create a new node struct Node* createNode(int data) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = data; newNode->prev = NULL;  newNode->next
= NULL;
   return newNode;
}

// Function to insert a node at the beginning of the list
void insertAtBeginning(struct Node** head_ref, int data) {
struct Node* newNode = createNode(data);     if
(*head_ref == NULL) {         *head_ref = newNode;
        return;
   }
   newNode->next = *head_ref;
   (*head_ref)->prev = newNode;
   *head_ref = newNode;
}

// Function to insert a node at the end of the list void
insertAtEnd(struct Node** head_ref, int data) {
struct Node* newNode = createNode(data);
struct Node* temp = *head_ref;     if (*head_ref ==
NULL) {        *head_ref = newNode;
```

```
        return;
    }
    while (temp->next != NULL) {
temp = temp->next;
    }
    temp->next = newNode;     newNode-
>prev = temp;
}

// Function to insert a node at a specific position
```

```
void insertAtPosition(struct Node** head_ref, int data, int position) {
if (position < 1) {        printf("Invalid position\n");        return;
   }
   if (position == 1) {
insertAtBeginning(head_ref, data);        return;
   }
   struct Node* newNode = createNode(data);
struct Node* temp = *head_ref;     for (int i = 1; i <
position - 1 && temp != NULL; i++) {        temp =
temp->next;
   }
   if (temp == NULL) { printf("Position
out of range\n");        return;
   }
   newNode->next = temp->next;
if (temp->next != NULL) {        temp->next->prev
= newNode;
   }
   temp->next = newNode;     newNode-
>prev = temp;
}

// Function to delete the first node void
deleteFirstNode(struct Node** head_ref) {     if
(*head_ref == NULL) { printf("List is empty\n"); return;
   struct Node* temp = *head_ref;
*head_ref = temp->next;     if
(*head_ref != NULL) {



   }
```

```
(*head_ref)->prev = NULL;
    }
    free(temp);
}

// Function to delete the last node void
deleteLastNode(struct Node** head_ref) {     if
(*head_ref == NULL) {         printf("List is empty\n");
return;
    }
    struct Node* temp = *head_ref;
while (temp->next != NULL) {         temp
= temp->next;
    }
    if (temp->prev != NULL) {         temp->prev-
>next = NULL;
    } else {
        *head_ref = NULL;
    }
    free(temp);
}

// Function to delete a node at a specific position void
deleteAtPosition(struct Node** head_ref, int position) {     if
(*head_ref == NULL) { printf("List is empty\n"); return;
    if (position < 1) {
        printf("Invalid position\n");
return;



    }
```

```
    }
    if (position == 1) {
deleteFirstNode(head_ref);         return;
    }
    struct Node* temp = *head_ref;     for (int i = 1; i
< position && temp != NULL; i++) {         temp =
temp->next;
                        }
    if (temp == NULL) {
printf("Position out of range\n");         return;
    }
    if (temp->next != NULL) {         temp->next-
>prev = temp->prev;
    }
    temp->prev->next = temp->next;
free(temp);
}

// Function to search for a node with a given value
struct Node* searchNode(struct Node* head, int key) {
struct Node* temp = head;     while (temp != NULL) {
if (temp->data == key) {             return temp;
        }
        temp = temp->next;






    }
```

```
    return NULL;
}

// Function to display the doubly linked list
void displayList(struct Node* head) {     struct
Node* temp = head;     while (temp
!= NULL) {         printf("%d ", temp->data);
temp = temp->next;
    }
printf("\n");
}
int main() {     struct Node* head =
NULL;     int choice, data,
position, key;
      do {         printf("\n1. Insert at
Beginning\n");         printf("2. Insert at
End\n");         printf("3. Insert at
Position\n");         printf("4. Delete First
Node\n");         printf("5. Delete Last
Node\n");         printf("6. Delete at
Position\n");         printf("7. Search for a
Node\n");         printf("8. Display
List\n");         printf("9. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

        switch(choice) {
case 1:
printf("Enter data to insert
at         beginning:         ");
scanf("%d",           &data);
insertAtBeginning(&head,
```

```
data);                break;
case 2:
          printf("Enter data to insert at end: ");
scanf("%d", &data);
insertAtEnd(&head, data);               break;          case
3:
          printf("Enter data to insert: ");
scanf("%d", &data);               printf("Enter position
to insert at: ");                scanf("%d", &position);
insertAtPosition(&head, data, position);
break;          case 4:
deleteFirstNode(&head);                break;          case
5:
          deleteLastNode(&head);
break;          case 6:
          printf("Enter position to delete: ");
scanf("%d", &position);
deleteAtPosition(&head, position);               break;
case 7:
          printf("Enter value to search: ");
scanf("%d", &key);               struct Node* result =
searchNode(head, key);               if (result != NULL) {
printf("%d found in the list.\n", key);               } else {
printf("%d not found in the list.\n", key);
          }
break;          case
8:
          printf("Doubly linked list: ");
displayList(head);               break;          case
9:
```

```
printf("Exiting...\n");                break;
default:                printf("Invalid
choice\n");
        }
    } while (choice != 9);

    return 0;
}
```

**Experiment 3:**

**APPLICATION OF SINGLY**

**LINKED LIST**

Algorithm:

1. Start
2. Define structure
3. Create term functions
4. Insert term into the polynomial and add, subtract and multiplication these polynomial and display it 5. End Program:

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

// Define structure for a term in
polynomial struct Term {    int coefficient;
int exponent;
struct Term *next;
};

typedef struct Term Term;

// Function to create a new term
Term *createTerm(int coeff, int exp) {    Term
*newTerm = (Term *)malloc(sizeof(Term));    if (newTerm
== NULL) {        printf("Memory
```

```
allocation failed\n");         exit(1);
    }
    newTerm->coefficient = coeff;
newTerm->exponent = exp;     newTerm-
>next = NULL;     return newTerm;
}

// Function to insert a term into the polynomial void
insertTerm(Term **poly, int coeff, int exp) {
Term *newTerm = createTerm(coeff, exp);     if
(*poly == NULL) {         *poly = newTerm;
    } else {
        Term *temp = *poly;
while (temp->next != NULL) {
temp = temp->next;
        }
        temp->next = newTerm;
    }
}

// Function to display the polynomial void
displayPolynomial(Term *poly) {     if
(poly == NULL) {         printf("Polynomial
is empty\n");
    } else {         while (poly != NULL) {
printf("(%dx^%d) ", poly->coefficient, poly->exponent);
poly = poly->next;             if (poly != NULL) {                 printf("+
");
            }         }
printf("\n");
    }
}
```

```
// Function to add two polynomials
Term *addPolynomials(Term *poly1, Term *poly2) {     Term
*result = NULL;     while (poly1 != NULL && poly2 != NULL) {
if (poly1->exponent > poly2->exponent) {
insertTerm(&result, poly1->coefficient, poly1->exponent);            poly1
= poly1->next;
} else if (poly1->exponent < poly2->exponent) {
insertTerm(&result, poly2->coefficient, poly2->exponent);            poly2
= poly2->next;
        } else {
            insertTerm(&result, poly1->coefficient + poly2->coefficient, poly1->exponent);
poly1 = poly1->next;            poly2 = poly2->next;
        }
    }
    while (poly1 != NULL) {        insertTerm(&result, poly1>coefficient,
poly1->exponent);        poly1 = poly1->next;
    }
    while (poly2 != NULL) {        insertTerm(&result, poly2>coefficient,
poly2->exponent);        poly2 = poly2->next;
    }
    return result;
}

// Function to subtract two polynomials
Term *subtractPolynomials(Term *poly1, Term *poly2) {     Term
*result = NULL;     while (poly1 != NULL && poly2 != NULL) {         if
(poly1->exponent > poly2->exponent) {
insertTerm(&result, poly1->coefficient, poly1->exponent);            poly1
= poly1->next;
        } else if (poly1->exponent < poly2->exponent) {
insertTerm(&result, -poly2->coefficient, poly2->exponent);            poly2
```

```
= poly2->next;
        } else {             insertTerm(&result, poly1->coefficient - poly2->coefficient,
poly1>exponent);             poly1 = poly1->next;             poly2 = poly2->next;
        }
    }
    while (poly1 != NULL) {        insertTerm(&result, poly1>coefficient,
poly1->exponent);        poly1 = poly1->next;
    }
    while (poly2 != NULL) {        insertTerm(&result, -poly2>coefficient,
poly2->exponent);        poly2 = poly2->next;
    }
    return result;
}

// Function to multiply two polynomials
Term *multiplyPolynomials(Term *poly1, Term *poly2) {
    Term *result = NULL;     Term *temp1 = poly1;     while (temp1 != NULL) {
Term *temp2 = poly2;        while (temp2 != NULL) {
insertTerm(&result, temp1->coefficient * temp2->coefficient, temp1-
>exponent + temp2->exponent); temp2
= temp2->next;
        }
        temp1 = temp1->next;
    }
    return result;
}

// Main function int main()
{
    Term *poly1 = NULL;
    Term *poly2 = NULL;
```

```
    // Insert terms for polynomial 1     insertTerm(&poly1,
5, 2);     insertTerm(&poly1,
-3, 1);     insertTerm(&poly1, 2, 0);

    // Insert terms for polynomial 2     insertTerm(&poly2,
4, 3);     insertTerm(&poly2,
2, 1);

    printf("Polynomial 1: ");
displayPolynomial(poly1);

    printf("Polynomial 2: ");
displayPolynomial(poly2);

    Term *sum = addPolynomials(poly1, poly2);     printf("Sum:
");     displayPolynomial(sum);

    Term *difference = subtractPolynomials(poly1, poly2);
printf("Difference: ");     displayPolynomial(difference);

    Term *product = multiplyPolynomials(poly1, poly2);
printf("Product: ");     displayPolynomial(product);

    return 0;
}
```

**Experiment 4:**

**IMPLEMENTATION OF**

**STACK USING ARRAY**

**AND LINKED LIST**

**IMPLEMENTATION**

Algorithm:

1.Start

2. Create a structure and functions for the given operations

3.Initialize stack array with capacity and top=-1
4.To push an element into a stack read the data to be pushed. If the top is equal to capacity-1 display stack overflow. Otherwise increment the top and push the data onto stack at index top

5.To pop an element from a stack if the top is equal to -1 display as stack underflow. Otherwise pop data from stack at index top the decrement the top and display the popped data

6. To return the top most element from a stack if the top is equal to -1 display stack is empty. Otherwise display data at index top

7.After these operations display all elements in stack from top to 0

8.End

Program:

```
#include <stdio.h>

#include <stdlib.h>


// Structure for node in linked list implementation

struct Node {     int data;     struct Node* next;

};


// Structure for stack using linked list implementation

struct StackLL {     struct Node* top;

};


// Structure for stack using array implementation

struct StackArray {     int* array;     int top;     int capacity;

};


// Function to initialize stack using linked list implementation

struct StackLL* createStackLL() {
    struct StackLL* stack = (struct StackLL*)malloc(sizeof(struct StackLL));

stack->top = NULL;     return stack;

}
```

```
// Function to initialize stack using array implementation struct StackArray*
createStackArray(int capacity) {     struct StackArray* stack = (struct
StackArray*)malloc(sizeof(struct StackArray));     stack->capacity = capacity;
stack->top = -1;     stack->array = (int*)malloc(stack->capacity * sizeof(int));     return
stack;
}

// Function to check if the stack is empty int
isEmptyLL(struct StackLL* stack) {     return
stack->top == NULL;
}

// Function to check if the stack is empty int
isEmptyArray(struct StackArray* stack) {     return
stack->top == -1;
}

// Function to push element into stack using linked list implementation
void pushLL(struct StackLL* stack, int data) {     struct Node* newNode
= (struct Node*)malloc(sizeof(struct Node));     newNode>data = data;
newNode->next = stack->top;     stack->top = newNode;
}

// Function to push element into stack using array implementation
void pushArray(struct StackArray* stack, int data) {     if (stack>top
== stack->capacity - 1) {         printf("Stack Overflow\n");
return;
    }
    stack->array[++stack->top] = data;
}
```

```
// Function to pop element from stack using linked list implementation
int popLL(struct StackLL* stack) {     if (isEmptyLL(stack)) {
printf("Stack Underflow\n");         return -1;
    }
    struct Node* temp = stack->top;
int data = temp->data;     stack>top
= stack->top->next;     free(temp);
return data;
}

// Function to pop element from stack using array implementation int
popArray(struct StackArray* stack) {     if
(isEmptyArray(stack)) {         printf("Stack Underflow\n");         return
-1;
    }
    return stack->array[stack->top--];
}

// Function to return top element from stack using linked list implementation int
peekLL(struct StackLL* stack) {     if (isEmptyLL(stack)) {         printf("Stack is
empty\n");         return -1;
    }
    return stack->top->data;
}

// Function to return top element from stack using array implementation
int peekArray(struct StackArray* stack) {     if (isEmptyArray(stack)) {
printf("Stack is empty\n");         return -1;
    }
    return stack->array[stack->top];
}
```

```
// Function to display elements in stack using linked list implementation void
displayLL(struct StackLL* stack) {     if (isEmptyLL(stack)) {
printf("Stack is empty\n");         return;
    }
    struct Node* temp = stack->top;
printf("Elements in stack: ");     while
(temp != NULL) {         printf("%d ",
temp->data);         temp = temp-
>next;
                                          }
printf("\n");
}

// Function to display elements in stack using array implementation void
displayArray(struct StackArray* stack) {     if
(isEmptyArray(stack)) {         printf("Stack is empty\n");         return;
    }
    printf("Elements in stack: ");
for (int i = stack->top; i >= 0; i--) {         printf("%d
", stack->array[i]);
    }
printf("\n");
}
int main() {
    // Test linked list implementation     struct
StackLL* stackLL = createStackLL();
pushLL(stackLL, 1);     pushLL(stackLL, 2);
pushLL(stackLL, 3);     displayLL(stackLL);
printf("Top element: %d\n", peekLL(stackLL));
printf("Popped element: %d\n", popLL(stackLL));
displayLL(stackLL);
```

```
    // Test array implementation     struct StackArray*
stackArray = createStackArray(5);
pushArray(stackArray, 4);     pushArray(stackArray, 5);
pushArray(stackArray, 6);     displayArray(stackArray);     printf("Top
element: %d\n", peekArray(stackArray));     printf("Popped element:
%d\n", popArray(stackArray));     displayArray(stackArray);


    return 0;
}
```

**Experiment**

**5:APPLICATION OF**

**STACK(INFIX TO**

**POSTFIX)** Algorithm:

1. Start
2. Initialize variables and stack
3. Read infix expression from the user
4. Scan infix expression by character
5. If the character is an operand append it to the postfix expression
6. If the character is an open parenthesis push it onto a stack
7. If the character is a closed parenthesis pop and append operators from the stack until an open parenthesis is encountered.
8. If the character is an operator while the stack is not empty and precedence of the top operator is greater or equal to precedence of the scanned operator, pop and append top operator to postfix expression 9. Push scanned operator onto the stack

10. After scanning the entire infix expression pop and append all the operators from the stack 11. Output the postfix expression

12. End Program:

```
#include <stdio.h>

#include <stdlib.h>
#include <string.h>

#include <ctype.h>


#define MAX_SIZE 100
```

```
// Structure for stack
struct Stack {     int top;
unsigned capacity;
char
*array;
};

// Function to create a stack of given capacity struct Stack*
createStack(unsigned capacity) {     struct Stack* stack = (struct Stack*)
malloc(sizeof(struct Stack));     stack->capacity = capacity;     stack-
>top = -1;     stack->array = (char*) malloc(stack->capacity *
sizeof(char));     return stack;
}

// Function to check if the stack is full int isFull(struct
Stack* stack) {     return stack-
>top == stack->capacity - 1;
}

// Function to check if the stack is empty
int isEmpty(struct Stack* stack) {     return
stack->top == -1;
}

// Function to push an item to the stack
void push(struct Stack* stack, char item)
{     if (isFull(stack))         return;
   stack->array[++stack->top] = item;
}

// Function to pop an item from the stack
char pop(struct Stack* stack) {     if
```

```
(isEmpty(stack))         return '\0';     return
stack->array[stack->top--];
}
// Function to get the precedence of operators
int precedence(char op) {     if (op
== '+' || op == '-')         return 1;     else if (op
== '*' || op == '/')         return 2;     else
return -1;
}
// Function to convert infix expression to postfix void
infixToPostfix(char* infix, char* postfix) {     struct
Stack* stack = createStack(strlen(infix));
int i, j;     for (i = 0, j = -1; infix[i]; ++i) { if
(isalnum(infix[i]))             postfix[++j] = infix[i];         else if (infix[i]
== '(')             push(stack, '(');         else if (infix[i] == ')') {
while (!isEmpty(stack) && stack->array[stack->top] != '(')
postfix[++j] = pop(stack);             if (!isEmpty(stack) && stack>array[stack-
>top] != '(')                 return; // Invalid expression             else
pop(stack);
        } else {
            while (!isEmpty(stack) && precedence(infix[i]) <= precedence(stack-
>array[stack->top]))
postfix[++j] = pop(stack);
push(stack, infix[i]);
        }
    }
    while (!isEmpty(stack))
postfix[++j] = pop(stack);     postfix[++j]
= '\0';
}
```

```
// Driver program int main()
{     char infix[MAX_SIZE];
char postfix[MAX_SIZE];

    printf("Enter an infix expression: ");
fgets(infix, MAX_SIZE, stdin);     infix[strcspn(infix,
"\n")] = 0;

    infixToPostfix(infix, postfix);

    printf("Postfix expression: %s\n", postfix);

    return 0;
}
```

**Experiment**

**6:EVALUATING**

**EXPRESSION.**

Algorithm:

1. Start
2. Create an empty stack to hold operands.
3. Initialize a variable top to -1 which represents the top of the stack
4. Read the input from user
5. Iterate through each character in the expression
6. If the character is a digit, convert the character to its integer value and push the integer into stack.
7. if the character is an operator, pop the top two operands from the stack and perform the corresponding operation.
8. Get the result and display it
9. End Program:

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

#define MAX_SIZE 100
```

```
int stack[MAX_SIZE]; int
top = -1;

void push(int item) {     if
(top >= MAX_SIZE - 1) {
printf("Stack Overflow\n");
} else {         top++;
stack[top] = item;
    }
}
int pop() {     if (top < 0) {
printf("Stack Underflow\n");
        return -1;     } else
{         return stack[top-
-];
    }
}
int evaluateExpression(char* exp) {
int i, operand1, operand2, result;     for
(i = 0; exp[i] != '\0'; i++) {         if
(isdigit(exp[i])) {             push(exp[i]
- '0');         } else {             operand2 =
pop();             operand1 = pop();
switch (exp[i]) {                 case '+':
                    push(operand1 + operand2);
break;                 case '-':
                    push(operand1 - operand2);                     break;
case '*':
                    push(operand1 * operand2);
break;                 case '/':
                    push(operand1 / operand2);
break;
            }
        }
    }
    result = pop();
return result;
}

int main() {     char
exp[MAX_SIZE];
    printf("Enter the arithmetic expression: ");
scanf("%s", exp);
    int result = evaluateExpression(exp);
printf("Result: %d\n", result);     return 0;
}
```

**Experiment 7:**
**IMPLEMENTATION OF QUEUE USING ARRAY AND LINKED LIST IMPLEMENTATION**

Algorithm:

1. Start
2. To enqueuer an element read the value
3. If rear is equal to MAX_SIZE-1, print Queue is full
4. Otherwise if front is -1, set front to 0 , increment rear by 1 and assign the value to queue[rear].
5. To dequeuer an element if front is -1 print Queue is emprty and return 1
6. Otherwise assign element as queue[front], increment front by 1
7. End

Program using array:

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

#define MAX_SIZE 100

int queue[MAX_SIZE]; int
front = -1, rear = -1;

void enqueue(int value);
int dequeue(); void
display();

int main() {
enqueue(10);     enqueue(20);
enqueue(30);

   display();

   dequeue();

   display();
      return
0;
}

void enqueue(int value) {     if
(rear == MAX_SIZE - 1) {
printf("Queue is full.\n");
} else {         if (front ==
-1) {            front = 0;
```

```
        }         rear++;          queue[rear]
= value;
    }
}

int dequeue() {     int element;
if (front == -1) {
printf("Queue is empty.\n");
return -1;     } else {          element
= queue[front];          front++;
if (front > rear) {
            front = rear = -1;
        }
        return element;
    }
}

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

Program using linked list:

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

struct Node {     int
data;     struct Node*
next;
};

struct Node* front = NULL;
struct Node* rear = NULL;

void enqueue(int value);
int dequeue(); void
display();
```

```
int main() {
enqueue(10);     enqueue(20);
enqueue(30);

    display();

    dequeue();

    display();

    return 0;
}

void enqueue(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));     newNode-
>data = value;      newNode-
>next = NULL;

    if (rear == NULL) {
front = rear = newNode;
    } else {
        rear->next = newNode;
        rear = newNode;
    }
}

int dequeue() {      if (front ==
NULL) {         printf("Queue is
empty.\n");
        return -1;
} else {         struct Node*
temp = front;
int element = temp->data;         front
= front->next;         free(temp);
if (front ==
NULL) {             rear = NULL;
       }
        return element;
    }
}

void display() {     if (front ==
NULL) {         printf("Queue is
empty.\n");
    } else {
```

```
        struct Node* temp = front;
printf("Queue elements: ");         while (temp
!= NULL) {             printf("%d ",
temp->data);
            temp = temp->next;
        }
        printf("\n");
    }
}
```

**Experiment 8**
**:PERFORMING TREE**
**TRAVERSAL**
**TECHNIQUES**

Algorithm:

1. Start
2. Create a node which contains data, left, right their member
3. Create 3 different types of functions to traversal in 3 different ways: inorder, preorder, postorder.
4. Call each functions and display the output
5. End

Program:

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

#define MAX_SIZE 100

int queue[MAX_SIZE]; int
front = -1, rear = -1;

void enqueue(int value);
int dequeue(); void display();

int main() {
enqueue(10);
enqueue(20);
enqueue(30);

    display();

   dequeue();

    display();
```

```
    return 0;
}

void enqueue(int value) {     if
(rear == MAX_SIZE - 1) {
printf("Queue is full.\n");
} else {         if (front ==
-1) {             front = 0;
        }        rear++;         queue[rear]
= value;
    }
}

int dequeue() {     int element;
if (front == -1) {
printf("Queue is empty.\n");
return -1;     } else {         element
= queue[front];         front++;
if (front > rear) {
            front = rear = -1;
        }
        return element;
    }
}

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

**Experiment 9:**
**IMPLEMENTATION OF**
**BINARY SEARCH**
**TREE** Algorithm:

1. Start
2. Defines a structure Node representing a node in the binary search tree. Each node contains data, left child pointer, and right child pointer.
3. Provides a  function to create a new node with the given value and initialize its pointers.

4. To insert a new node into the binary search tree while maintaining the BST property. Create a function to check if the value is less than the current node's data, it traverses to the left subtree; otherwise, it traverses to the right subtree.
5. To delete a node from the binary search tree while preserving the BST property. Create a function to handle cases where the node has zero, one, or two children by finding the successor node and replacing the node to be deleted with it.
6. Create a function to find the node with the minimum value in a subtree, which is used in deletion operation.
7. To search for a value in the binary search tree, Create a recursive function to traverses the tree, comparing the value with each node's data until the value is found or the tree is exhausted.
8. Provide a function to perform an inorder traversal of the binary search tree, printing the nodes in sorted order.
9. End Program:

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

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

struct Node* createNode(int
value) {     struct Node* newNode
= (struct
Node*)malloc(sizeof(struct  Node));
          newNode->data =  value;
   newNode->left = NULL;
newNode->right = NULL;
   return newNode;
}

struct Node* insert(struct
Node* root, int value) {     if
(root == NULL) {
return  createNode(value);
   }

   if (value < root->data) {
      root->left =
insert(root->left, value);
} else if (value > root>data)
{        root>right =
insert(root-
```

```
>right, value);
    }

    return root;
}

struct Node*
minValueNode(struct
Node* node) {     struct
Node* current =  node;

    while (current && current->left
!= NULL) {
current = current>left;
    }

    return current;
}

struct Node* deleteNode(struct
Node* root, int value) {
if (root == NULL) {
        return root;
    }

    if (value < root->data)
{         root->left =
deleteNode(root->left,
value);     } else if (value
> root>data) {
root>right =
deleteNode(root-
>right, value);     } else {
if (root->left ==  NULL) {
struct Node* temp = root-
>right;             free(root);
return temp;
        } else if (root->right
== NULL) {              struct
Node* temp = root->left;
free(root);             return
temp;
        }

        struct Node* temp = minValueNode(root-
>right);         root->data = temp>data;
```

```
        root->right = deleteNode(root>right,
  temp->data);
     }

     return root;
  }

  struct Node* search(struct
  Node* root, int value) {     if
  (root == NULL || root>data
  == value) {
  return root;
     }

     if (root->data < value) {
  return search(root- >right,
  value);
     }

     return search(root->left,
  value);  }

  void display(struct Node*
  root) {     if (root != NULL)
  {         display(root->left);
  printf("%d ", root>data);
  display(root->right);
     }
  }

  int main() {     struct
  Node* root =
  NULL;
     root = insert(root, 50);
insert(root, 30);     insert(root,
    20);     insert(root, 40);     insert(root,
70);     insert(root,
    60);     insert(root, 80);

     printf("Binary Search Tree
  Inorder Traversal: ");
  display(root);
     printf("\n");

     root = deleteNode(root,
  20);     printf("Binary
```

```
Search Tree Inorder
Traversal after deleting 20:
");     display(root);
printf("\n");

   struct Node*
searchResult = search(root,
30);
   if (searchResult != NULL)
{
      printf("Element 30
found in the Binary
Search Tree.\n");
   } else {
printf("Element 30 not found
in the Binary
Search Tree.\n");
   }

   return 0;
}
```

**Experiment 10:**
**IMPLEMENTATION OF**
**AVL TREE** Algorithm:

1. Start
2. Defines a structure Node representing a node in the AVL tree, containing data, left and right child pointers, and height.
3. Create a function to calculate the height of a node and its balance factor.
4. Implement a function to create a new node with the given data and initial height.
5. Create a function to performs a right rotation to balance the tree.
6. Create a function to performs left rotation to balance the tree.
7. Implement a function to insert a node into the AVL tree while maintaining AVL property and performing rotations as needed.
8. For deletion, implements a function to find the node with the minimum value in a subtree and implements a function to delete a node from the AVL tree while maintaining AVL property and performing rotations as needed.
9. Provides a function to perform inorder traversal of the AVL tree, printing the nodes in sorted order
10. End

Program:

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

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

// Function to get the height of a node int height(Node
*node) {
    if (node == NULL)
return 0;     return
node->height;
}
// Function to get the balance factor of a node int
balance_factor(Node *node) {     if (node
== NULL)         return 0;     return height(node-
>left) - height(node->right);
}

// Function to create a new node
Node* newNode(int data) {
    Node* node = (Node*)malloc(sizeof(Node));     node-
>data = data;     node->left = NULL;     node->right =
NULL;     node->height = 1;
    return node;
}

// Function to perform a right rotation
Node* rotate_right(Node *y) {
   Node *x = y->left;
   Node *T2 = x->right;

   // Perform rotation   x>right
= y; y->left = T2;     // Update
heights    y->height = 1 +
(height(y->left) > height(y-
>right) ? height(y->left) :
height(y-
>right));     x->height = 1 + (height(x->left) > height(x->right) ?
height(x->left) :
height(x>right));

    return x;
```

```
}

// Function to perform a left rotation
Node* rotate_left(Node *x) {
   Node *y = x->right;
   Node *T2 = y->left;

   // Perform rotation     y-
>left = x;      x-
>right = T2;

   // Update heights
   x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x-
>right));      y->height = 1 + (height(y->left) > height(y->right) ?
height(y->left) :
height(y>right));

   return y;
}

// Function to insert a node into AVL tree
Node* insert(Node *node, int data) {     if
(node == NULL)
return newNode(data);

   if (data < node->data)
      node->left = insert(node->left, data);
else if (data > node->data)
      node->right = insert(node->right, data);
else // Duplicate keys not allowed
      return node;

   // Update height of current node     node->height = 1 + (height(node->left) >
height(node->right) ? height(node->left) : height(node->right));

  // Get the balance factor    int
balance = balance_factor(node);

  // Perform rotations if needed     if (balance
> 1 && data < node->left->data)      return
rotate_right(node);
  if (balance < -1 && data > node->right->data)
return rotate_left(node);    if (balance > 1 && data
> node->left->data) {        node->left =
rotate_left(node->left);
      return rotate_right(node);
```

```
    }
    if (balance < -1 && data < node->right->data) {
node->right = rotate_right(node->right);         return rotate_left(node);
    }

    return node;
}

// Function to find the node with minimum value
Node* minValueNode(Node *node) {
Node* current = node;     while
(current->left != NULL)         current
= current->left;     return current;
}
// Function to delete a node from AVL tree Node*
deleteNode(Node *root, int data) {
    if (root == NULL)
        return root;

    if (data < root->data)
        root->left = deleteNode(root->left, data);
else if (data > root->data)
        root->right = deleteNode(root->right, data);
else {
        if (root->left == NULL || root->right == NULL) {
Node *temp = root->left ? root->left : root->right;

            if (temp == NULL) {
temp = root;                  root
= NULL;
            } else
                *root = *temp; // Copy the contents of the non-empty child

            free(temp);
        } else {
            Node *temp = minValueNode(root->right);            root->data
 = temp->data;
           root->right = deleteNode(root->right, temp->data);
       }
   }
    if (root == NULL)
       return root;

    // Update height of current node    root->height = 1 + (height(root->left) >
height(root->right) ? height(root->left) : height(root->right));
```

```
    // Get the balance factor      int
balance = balance_factor(root);

    // Perform rotations if needed
    if (balance > 1 && balance_factor(root->left) >= 0)
return rotate_right(root);
    if (balance > 1 && balance_factor(root->left) < 0) {
        root->left = rotate_left(root->left);
return rotate_right(root);
    }
    if (balance < -1 && balance_factor(root->right) <= 0)
return rotate_left(root);
    if (balance < -1 && balance_factor(root->right) > 0) {          root-
>right = rotate_right(root->right);
return rotate_left(root);
    }

    return root;
}

// Function to print AVL tree inorder
void inorder(Node *root) {     if (root !=
NULL) {         inorder(root->left);
printf("%d ", root->data);
        inorder(root->right);
    }
}

int main() {
    Node *root = NULL;

    // Inserting nodes     root
= insert(root, 10);     root =
insert(root, 20);     root =
insert(root, 30);     root =
insert(root, 40);     root =
insert(root, 50);    root =
insert(root, 25);

   printf("Inorder traversal of the constructed AVL tree: ");
inorder(root);    printf("\n");
   // Deleting node     printf("Delete node
30\n");       root  =  deleteNode(root,  30);
printf("Inorder traversal after deletion: ");
    inorder(root);
    printf("\n");
```

```
    return 0;
}
```

**Experiment 11: IMPLEMENTATION OF BFS,DFS** Algorithm:

1. Start
2. Create a node which contains vertex and next as their members.
3. Allocates memory dynamically for nodes and the graph structure using malloc().
4. Create a graph with a specified number of vertices and initialize adjacency lists.
5. Create a function to add the edges between the vertices
6. Performs BFS traversal starting from a given vertex using a queue data structure to maintain order.
7. Conducts DFS traversal starting from a specified vertex, employing recursion to explore graph branches.

8.Displays the adjacency list representation of the graph and the traversal sequences for both BFS and DFS.
9. End

Program:

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

#define MAX 100

struct Node {     int
vertex;     struct Node*
next;
};

struct Node* createNode(int v);

struct Graph {     int
numVertices;     struct
Node** adjLists;
int* visited;
};

struct Graph* createGraph(int vertices);

void addEdge(struct Graph* graph, int src, int dest);  void
printGraph(struct Graph* graph);
```

```
void BFS(struct Graph* graph, int startVertex);

void DFS(struct Graph* graph, int startVertex);

int main() {
    struct Graph* graph = createGraph(4);
addEdge(graph, 0, 1);     addEdge(graph, 0,
2);     addEdge(graph, 1, 2);
addEdge(graph, 2, 0);     addEdge(graph,
2, 3);     addEdge(graph, 3, 3);

    printf("Graph:\n");
    printGraph(graph);

    printf("\nBFS Traversal:\n");
    BFS(graph, 2);

    printf("\nDFS Traversal:\n");
    DFS(graph, 2);

    return 0;
}

struct Node* createNode(int v) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->vertex = v;     newNode->next = NULL;
    return newNode;
}

struct Graph* createGraph(int vertices) {      struct Graph* graph = (struct
Graph*)malloc(sizeof(struct Graph));     graph>numVertices = vertices;

    graph->adjLists = (struct Node**)malloc(vertices * sizeof(struct Node*));
graph>visited = (int*)malloc(vertices * sizeof(int));

    for (int i = 0; i < vertices; i++) {         graph-
>adjLists[i] = NULL;          graph-
>visited[i] = 0;
    }

    return graph;
}
void addEdge(struct Graph* graph, int src, int dest) {
struct Node* newNode = createNode(dest);     newNode-
>next = graph->adjLists[src];
    graph->adjLists[src] = newNode;
```

```
    newNode = createNode(src);     newNode>next
= graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

void printGraph(struct Graph* graph) {     for
(int v = 0; v < graph->numVertices; v++) {
struct Node* temp = graph->adjLists[v];
printf("Vertex %d: ", v);
while (temp) {             printf("%d -> ",
temp->vertex);
            temp = temp->next;
        }
        printf("NULL\n");
    }
}

void BFS(struct Graph* graph, int startVertex) {
struct Node* queue[MAX];      int front =
0, rear = 0;     queue[rear] =
createNode(startVertex);
    graph->visited[startVertex] = 1;

    printf("Visited %d\n", startVertex);

    while (front <= rear) {
        struct   Node*   currentNode   =   queue[front];
front++;            while  (currentNode)  {                int
adjVertex = currentNode->vertex;            if (!graph-
>visited[adjVertex]) {              printf("Visited %d\n",
adjVertex);
queue[++rear] = createNode(adjVertex);                 graph-
>visited[adjVertex] = 1;
            }
            currentNode = currentNode->next;
        }
    }
}

void DFSUtil(struct Graph* graph, int vertex) {
struct Node* temp = graph->adjLists[vertex];     graph-
>visited[vertex] = 1;     printf("Visited
%d\n", vertex);      while (temp) {
int adjVertex
= temp->vertex;         if (!graph-
```

```
>visited[adjVertex]) {
            DFSUtil(graph, adjVertex);
        }
        temp = temp->next;
    }
}

void DFS(struct Graph* graph, int startVertex) {     graph-
>visited[startVertex] = 1;
printf("Visited %d\n", startVertex);

    struct Node* temp = graph->adjLists[startVertex];

    while (temp) {         int adjVertex
= temp->vertex;         if (!graph-
>visited[adjVertex]) {
            DFSUtil(graph, adjVertex);
        }
        temp = temp->next;
    }
}
```

**Experiment 12:PERFORMING TOPOLOGICAL SORTING** Algorithm:

1. Start
2. Read the input from the user to create an adjacency matrix representing the graph.
3. Implements the DFS algorithm to traverse the graph recursively.
4. Create a recursive function to explores the graph starting from a given vertex and stops when all adjacent vertices have been visited.
5. After traversal, the program prints the ordering of vertices.
6. End

Program:

```
#include <stdio.h>

#define MAX_VERTICES 10

int graph[MAX_VERTICES][MAX_VERTICES] = {0};
int visited[MAX_VERTICES] = {0}; int
vertices;
```

```
void createGraph() {
    int i, j;
    printf("Enter the number of vertices: ");
scanf("%d", &vertices);
   printf("Enter the adjacency matrix:\n");  for
    (i = 0; i < vertices; i++) {
     for (j = 0; j < vertices; j++) {
     scanf("%d", &graph[i][j]);
        }
    }
}
void dfs(int vertex) {
    int i;
    printf("%d ", vertex);
visited[vertex] = 1;     for (i = 0; i < vertices;
i++) {         if (graph[vertex][i]
&& !visited[i]) {             dfs(i);
        }
    }
}

int main() {
    int i;
    createGraph();
    printf("Ordering of vertices after DFS traversal:\n");
    for (i = 0; i < vertices; i++) {
if (!visited[i]) {
dfs(i);
        }
    }
    return 0;
}
```

**Experiment 13 :**
**IMPLEMENTATION OF**
**PRIM’S ALGORITHM**

Algorithm:

1. Start
2. Input the number of vertices and the adjacency matrix representing the graph.
3.Find the vertex with the minimum key value among the vertices not yet included in the MST.
4. Initializes key values and mstset for all vertices , then iteratively select the vertex with the minimum key value and  updates the key values of its adjacent vertices of a shorter edge is found.
5.Print the edges of the MST along with their weights.

6. End

Program:

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

#define MAX_VERTICES 10
#define INF 999999

int graph[MAX_VERTICES][MAX_VERTICES]; int
vertices;

void createGraph() {
    int i, j;
    printf("Enter the number of vertices: ");
scanf("%d", &vertices);
    printf("Enter the adjacency matrix:\n");
for (i = 0; i < vertices; i++) {
for (j = 0; j < vertices; j++) {
scanf("%d", &graph[i][j]);
        }
    }
}

int findMinKey(int key[], bool mstSet[]) {
int min = INF, min_index;     for (int v = 0; v
< vertices; v++) {         if (mstSet[v] == false
&& key[v] < min) {
min = key[v];
            min_index = v;
        }
    }
    return min_index;
}

void printMST(int parent[]) {
printf("Edge \tWeight\n");     for (int i = 1; i < vertices; i++) {
printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
    }
}

void primMST() {
int parent[vertices];     int
key[vertices];
    bool mstSet[vertices];
```

```
    for (int i = 0; i < vertices; i++) {
key[i] = INF;
        mstSet[i] = false;
    }

    key[0] = 0; // Make key 0 so that this vertex is picked as the first vertex
parent[0] = -1; // First node is always root of MST

   for (int count = 0; count < vertices - 1; count++) {
int u = findMinKey(key, mstSet);
    mstSet[u] = true;
        for (int v = 0; v < vertices; v++) {
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v]) {
    parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }
    printMST(parent);
}

int main() {     createGraph();
primMST();
    return 0;
}
```

**Experiment 14 :**
**IMPLEMENTATION OF**
**DIJKSTRA'S**
**ALGORITHM**

Algorithm:

1. Start
2. The main function calls create graph funcion.
3. Input the number of vertices and the adjacency matrix representing the graph.
4.find the vertex ward the minimum distance from the source vertex among the vertices.
5. Then at implements Dijkstra's algorithm, it initializes distance values and sptset for all vertices, then iteratively updates the distance values unit all vertices are included in the shortest part pree.
6.Display the Output
7. End

Program:

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

#define MAX_VERTICES 10
#define INF 999999

int graph[MAX_VERTICES][MAX_VERTICES]; int
vertices;

void createGraph() {
    int i, j;
    printf("Enter the number of vertices: ");
scanf("%d", &vertices);      printf("Enter
the adjacency matrix:\n");      for (i = 0; i
< vertices; i++) {         for (j = 0; j <
vertices; j++) {
scanf("%d", &graph[i][j]);
        }
    }
}
int minDistance(int dist[], bool sptSet[]) {
int min = INF, min_index;     for (int v = 0; v <
vertices; v++) {         if (sptSet[v] == false &&
dist[v] <= min) {             min = dist[v];
min_index = v;
        }
    }
    return min_index;
}

void printSolution(int dist[]) {     printf("Vertex
\t Distance from Source\n");      for (int i = 0;
i < vertices; i++) {
        printf("%d \t %d\n", i, dist[i]);
    }
}

void dijkstra(int src) {
int dist[vertices];
    bool sptSet[vertices];

    for (int i = 0; i < vertices; i++) {
dist[i] = INF;
        sptSet[i] = false;
    }
```

```
    dist[src] = 0;

    for (int count = 0; count < vertices - 1; count++) {
int u = minDistance(dist, sptSet);
        sptSet[u] = true;

        for (int v = 0; v < vertices; v++) {
            if (!sptSet[v] && graph[u][v] && dist[u] != INF && dist[u] + graph[u][v] < dist[v])
{
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }

    printSolution(dist);
}

int main() {  createGraph();
     int source;
     printf("Enter the source vertex: ");
     scanf("%d", &source); dijkstra(source);
   return 0;
}
```

**Experiment**

**15:PROGRAM TO PERFORM**

**SORTING**

Algorithm:

1. Start
2. Read the number of elements n from the user
3. Read n elements into array
4. Sort the array using Quick sort function  and print the sorted array
5. sort the array using merge sort function and print the sorted array  6. End

Program:

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

void swap(int *a, int *b) {
int temp = *a;     *a = *b;
    *b = temp;
```

```
}

int partition(int arr[], int low, int high) {
int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)
{          if (arr[j] < pivot) {              i++;
             swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
void merge(int arr[], int l, int m, int r) {
int i, j, k;     int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
L[i] = arr[l + i];     for (j = 0;
j < n2; j++)         R[j] =
arr[m + 1 + j];
i = 0;     j = 0;
    k = l;     while (i < n1 &&
j < n2) {         if (L[i] <= R[j])
{             arr[k] = L[i];
i++;         } else {
arr[k] = R[j];             j++;
}         k++;
    }

    while (i < n1) {
arr[k] = L[i];
i++;         k++;
    }
```

```
    while (j < n2) {
arr[k] = R[j];
j++;         k++;
    }
}

void mergeSort(int arr[], int l, int r) {
if (l < r) {         int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);     merge(arr, l, m, r);
    }
}
int main() {
    int n;
    printf("Enter the number of elements: ");
scanf("%d", &n);

    int arr[n];
    printf("Enter %d elements:\n", n);
for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    printf("\nSorting using Quick Sort:\n");     quickSort(arr,
0, n - 1);     for (int i = 0; i
< n; i++) {         printf("%d ", arr[i]);
    }

    printf("\n\nSorting using Merge Sort:\n");
mergeSort(arr, 0, n - 1);     for (int i = 0; i <
n; i++) {         printf("%d ", arr[i]);
    }

    return 0;
}
```

**Experiment 16:**

**IMPLEMENTATION OF**

**COLLISION**

**RESOLUTION TECHNIQUES.**

Algorithm:

1. Start

2. Read data from the user
3. Allocate memory fir a new node
4. Set the data field of the new node to the input data
5. Set the next pointer of the new node to the NULL and return the pointer to the new node
6. Read key from the user
7. Calculate the index by taking the modulo of the key with table size and return the calculated index
8. Input the table and calculate the index using the hash function
9. Iterate until an empty slot is found in the table
10. Create a news node to the table at the calculated index and return the pointer to the inserted node
11. Assign the new miss to the table at the calculated index and return the pointer to the inserted node.
12. Repeat this to insert elements using closed addressing and rehashing
13. Then display the hash table using display function
14. End

Program:

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

#define TABLE_SIZE 10

typedef struct Node {
int data;     struct Node*
next;
} Node;

Node* createNode(int data) {
   Node* newNode = (Node*)malloc(sizeof(Node));
if (newNode == NULL) {        printf("Memory
allocation failed!\n");
      exit(1);
   }
   newNode->data = data;     newNode-
>next = NULL;
return newNode;
}

int hashFunction(int key) {
   return key % TABLE_SIZE;
}
```

```
Node* insertOpenAddressing(Node* table[], int key) {     int
index = hashFunction(key);     while (table[index]
!= NULL) {          index = (index + 1) %
TABLE_SIZE;
    }
    table[index] = createNode(key);
return table[index];
}

void displayHashTable(Node* table[]) {
printf("Hash Table:\n");     for (int i = 0; i
< TABLE_SIZE; i++) {
        printf("%d: ", i);         Node*
current = table[i];         while (current
!= NULL) {             printf("%d ",
current->data);
            current = current->next;
        }
printf("\n");
    }
}

Node* insertClosedAddressing(Node* table[], int key)
{     int index = hashFunction(key);     if (table[index]
== NULL) {          table[index] = createNode(key);
    } else {
        Node* newNode = createNode(key);         newNode-
>next = table[index];
table[index] = newNode;
    }
    return table[index];
}

int rehashFunction(int key, int attempt) {
// Double Hashing Technique      return (hashFunction(key) + attempt * (7
- (key % 7))) % TABLE_SIZE;
}

Node* insertRehashing(Node* table[], int key)
{     int index = hashFunction(key);     int
attempt = 0;     while (table[index] != NULL) {
attempt++;
        index = rehashFunction(key, attempt);
    }
    table[index] = createNode(key);
    return table[index];
```

```
}

int main() {
    Node* openAddressingTable[TABLE_SIZE] = {NULL};
    Node* closedAddressingTable[TABLE_SIZE] = {NULL};
    Node* rehashingTable[TABLE_SIZE] = {NULL};

    // Insert elements into hash tables
insertOpenAddressing(openAddressingTable, 10);
insertOpenAddressing(openAddressingTable, 20);
    insertOpenAddressing(openAddressingTable, 5);

    insertClosedAddressing(closedAddressingTable, 10);
insertClosedAddressing(closedAddressingTable, 20);
insertClosedAddressing(closedAddressingTable, 5);

    insertRehashing(rehashingTable, 10);
insertRehashing(rehashingTable, 20);
    insertRehashing(rehashingTable, 5);

    // Display hash tables
    displayHashTable(openAddressingTable);
displayHashTable(closedAddressingTable);
    displayHashTable(rehashingTable);

    return 0;
}
```