

# Experiment No. 1

**Aim:** To implement stack ADT using an array.

## Theory:

- **Pop():** Removing a data element from the stack data structure is called a pop operation. The pop operation has two following steps: The value of the top variable will be incremented by one whenever you delete an item from the stack.
- **Push():** It means insertion and deletion in the stack data structure can be done only from the top of the stack. You can access only the top of the stack at any given point in time. Inserting a new element in the stack is termed a push.
- **Peek():** In computer science, peek is an operation on certain abstract data types, specifically sequential collections such as stacks and queues, which returns the value of the top ("front") of the collection without removing the element from the collection.
- **Display():** You display the contents of a data structure or its subfields as you would any standalone field. You simply use the data structure name after EVAL to see the entire contents, or the subfield name to see a subset.

## Program :

```
#include<stdio.h>
#define N 5
int stack[N];
int top=-1;
void push()
{
    int x;
    printf("Enter data:");
    scanf("%d",&x);
    if(top==N-1)
    {
        printf("Overflow");
    }
    else
    {
        top++;
        stack[top]=x;
    }
}

void pop()
{
    int item;
    if(top== -1)
    {
        printf("Underflow");
    }
    else
    {
        item=stack[top];
        top--;
        printf("%d\n",item);
    }
}

void peek()
{
    if(top== -1)
    {
        printf("Stack is empty");
    }
}
```

```

}
else
{
printf("%d\n",stack[top]);
}
}

void display()
{
int i;
for(i=top;i>=0;i--)
{
printf("%d\t",stack[i]);
}
}

void main()
{
int ch;
do
{
printf("Enter choice for operation: ");
scanf("%d",&ch);
switch(ch)
{
case 1:
push();
break;
case 2:
pop();
break;
case 3:
peek();
break;
case 4:
display();
break;
default:
printf("Invalid choice");
}
}while(ch!=0);
}
}

```

**Output:**

```
C:\STORBUO3\BIN>TC
Enter choice for operation: 1
Enter data:2
Enter choice for operation: 1
Enter data:3
Enter choice for operation: 1
Enter data:4
Enter choice for operation: 2
4
Enter choice for operation: 1
Enter data:5
Enter choice for operation: 3
5
Enter choice for operation: 4
5      3      2      Enter choice for operation: s
```

**Conclusion :** This program demonstrates basic stack operations (push, pop, peek, display) using an array, with overflow and underflow checks. It allows user interaction through a menu-driven approach until the exit condition is met.

## Experiment No. 2

**Aim:** To Convert infix expression to postfix expression using stack ADT.

### Theory:

This program converts an infix expression (where operators are placed between operands, like 'A + B') into a postfix expression (where operators follow operands, like 'A B +') using a stack-based approach .

The key steps involved in this conversion process are:

- 1. Operand Handling:** If the current character is an operand (a letter or digit), it is directly added to the output (postfix expression).
- 2. Operator Handling:** If the character is an operator, the program compares its precedence with the operator at the top of the stack:
  - If the current operator has lower or equal precedence, it pops operators from the stack and adds them to the output until it finds an operator with lower precedence or an open parenthesis.
  - Then, the current operator is pushed onto the stack.
- 3. Parentheses Handling:**
  - When encountering an open parenthesis '(', it is pushed onto the stack.
  - When a closing parenthesis ')' is found, operators are popped from the stack and added to the output until an open parenthesis is found, which is then discarded.
- 4. End of Expression:** After the entire expression is scanned, any remaining operators in the stack are popped and added to the output.

This algorithm ensures that the resulting postfix expression is in the correct order for evaluation without needing parentheses, while respecting operator precedence and associativity. The stack plays a crucial role in temporarily holding operators and ensuring the proper sequencing of operations.

### Program:

```
#include<stdio.h>
#include<ctype.h>
char stack[100];
int top = -1;

void push(char x)
{
    stack[++top] = x;
}
```

```

char pop()
{
if(top == -1)
return -1;
else
return stack[top--];
}
int priority(char x)
{
if(x == '(')
return 0;
if(x == '+' || x == '-')
return 1;
if(x == '*' || x == '/')
return 2;
return 0;
}
int main()
{
char exp[100];
char *e, x;
printf("Enter the expression : ");
scanf("%s",exp);
printf("\n");

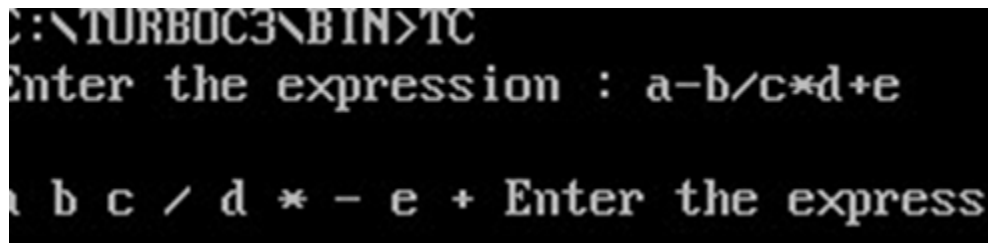
```

```

e = exp;
while(*e != '\0')
{
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:



```

C:\TURBOC3\BIN>TC
Enter the expression : a-b/c*d+e
a b c / d * - e + Enter the express

```

**Conclusion:** This program converts an infix expression to its postfix form using a stack for operator precedence handling. It processes the expression by pushing operators onto the stack and printing operands directly, ensuring proper order based on operator precedence and parentheses.

## Experiment No. 3

**Aim:** Evaluate Postfix expression using stack ADT

**Theory:** To evaluate a postfix expression using a stack ADT involves iterating through each symbol in the expression and performing operations based on whether the symbol is an operand or an operator. Here's a concise explanation of how to achieve this:

Steps to Evaluate a Postfix Expression:

- 1. Initialize a Stack:** Use a stack to hold operands during the evaluation process.
- 2. Iterate Through the Expression:** Start from the beginning of the postfix expression and process each symbol.
- 3. Operand Handling:**
  - If the symbol is an operand (number), push it onto the stack.
- 4. Operator Handling:**
  - If the symbol is an operator (+, -, \*, /, etc.), pop the required number of operands from the stack, perform the operation, and push the result back onto the stack.
- 5. Final Result:**
  - After processing all symbols in the postfix expression, the stack should contain only one element, which is the result of the evaluation.
- 6. Initialization:** The Stack class is used to manage operands during the evaluation.
- 7. Iterating through the Expression:** Each character (char) in the postfix expression is processed in sequence.
- 8. Operand Handling:** If char is a digit, it is converted to an integer and pushed onto the stack.
- 9. Operator Handling:** When encountering an operator (+, -, \*, /), operands are popped from the stack, the operation is performed, and the result is pushed back onto the stack.
- 10. Final Result:** After processing all characters in the expression, the stack will contain the final result of the postfix expression evaluation.

This method ensures that the postfix expression is evaluated correctly according to the postfix evaluation rules, using a stack ADT to manage the operands and perform the necessary operations efficiently. Adjustments may be needed based on specific requirements such as handling floating-point operations or more complex expressions.

### Program:

```
#include <stdio.h>
#include <ctype.h>
int stack[20];
int top = -1;
void push(int x) {
    stack[++top] = x;
}

int pop() {
    return stack[top--];
}

int main() {
    char exp[20];
    char *e;
    int n1, n2, n3, num;
```

```

printf("Enter the expression :: ");
scanf("%s", exp);
e = exp;
while (*e != '\0') {
if (isdigit(*e)) {
num = *e - '0'; // Convert character digit to
integer
push(num);
} else {
n1 = pop();
n2 = pop();
switch (*e) {
case '+':
n3 = n2 + n1; // Addition
break;
case '-':
n3 = n2 - n1; // Subtraction

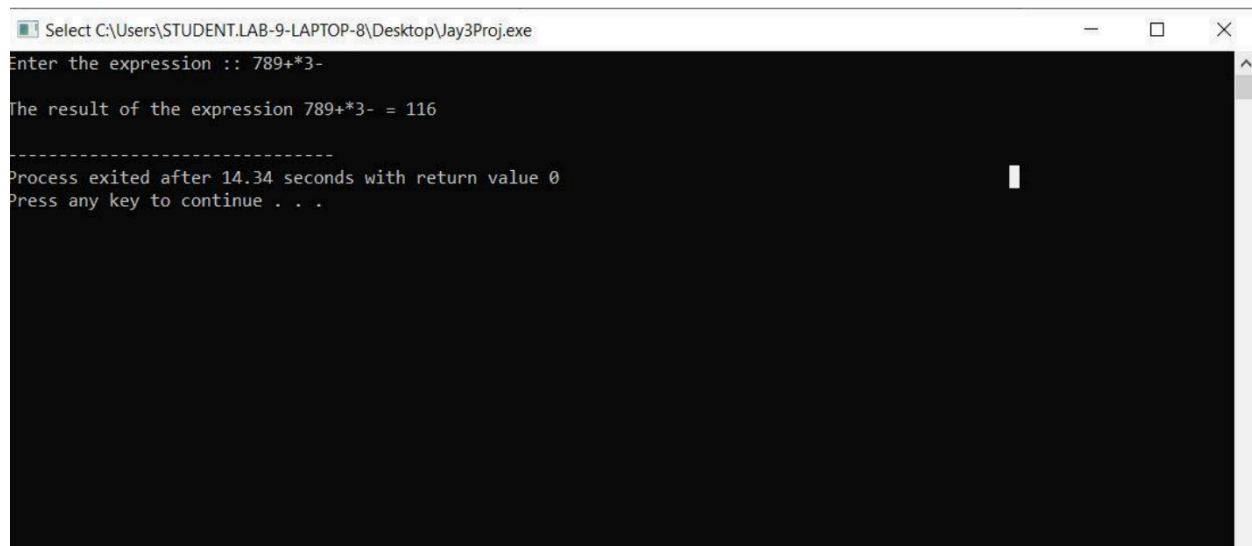
```

```

break;
case '*':
n3 = n2 * n1; // Multiplication
break;
case '/':
n3 = n2 / n1; // Division (n2 / n1, not n1 /
n2)
break;
}
push(n3);
}
e++;
}
printf("\nThe result of the expression %s =
%d\n", exp, pop());
return 0;
}

```

## Output:



```

Select C:\Users\STUDENT.LAB-9-LAPTOP-8\Desktop\Jay3Proj.exe
Enter the expression :: 789+*3-
The result of the expression 789+*3- = 116
-----
Process exited after 14.34 seconds with return value 0
Press any key to continue . . .

```

**Conclusion:** This program evaluates a postfix expression using a stack. It processes each character of the expression, pushing operands onto the stack and applying operators by popping operands, performing the operation, and pushing the result back onto the stack. The final result is the only value left on the stack after all operations are processed.

# Experiment No. 4

**Aim:** To implement a queue using array ADT.

**Theory:** A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle, meaning the first element added to the queue will be the first to be removed. 1. Array-Based Queue

In this program, a fixed-size array (queue[N] where  $N=5$ ) is used to represent the queue. Two important variables are maintained:

front : Points to the first element in the queue, i.e., the element that is dequeued first.

rear : Points to the last element in the queue, i.e., where the next element will be inserted.

Initially, both front and rear are set to -1 to represent an empty queue. The queue is dynamic within its fixed size, growing as elements are added (enqueue) and shrinking as elements are removed (dequeue). The queue size is limited by the value  $N$ , which means that at any point, no more than  $N$  elements can be present in the queue.

## 2. Queue Operations

### a. Enqueue Operation

The enqueue() function inserts a new element at the rear of the queue. The conditions for enqueueing are:

If  $\text{rear} == N-1$ , the queue is full (overflow condition), and no further elements can be added.

If the queue is initially empty ( $\text{front} == -1$  and  $\text{rear} == -1$ ), both front and rear are set to 0, indicating that the queue is now active and the first element is inserted.

If the queue has space ( $\text{rear} < N-1$ ), the rear pointer is incremented, and the new element is inserted at the updated position.

### b. Dequeue Operation

The dequeue() function removes the element from the front of the queue. The conditions for dequeuing are:

If  $\text{front} == -1$  and  $\text{rear} == -1$ , the queue is empty (underflow condition), and no elements can be removed.

If there is only one element left ( $\text{front} == \text{rear}$ ), removing that element will reset both front and rear to -1, marking the queue as empty.

For other cases, the front pointer is incremented to point to the next element, effectively removing the previous front element.

### c. Peek Operation

The peek() function allows the user to view the element at the front of the queue without removing it. If the queue is empty, it displays an appropriate message. Otherwise, it simply prints the element at the position indicated by front.

#### d. Display Operation

The display() function prints all elements in the queue in the order they were inserted. If the queue is empty, it shows a message indicating this. Otherwise, it iterates from front to rear and prints each element in sequence.

#### Program:

```
#include<stdio.h>
#define N 5
int queue [N];
int front=-1, rear=-1;

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

void peek()
{
    if(front== -1 && rear== -1)
    {
        printf("Queue is empty");
    }
    else
    {
        printf("%d", queue[front]);
    }
}

void enqueue(int x)
{
    if(rear==N-1)
    {
        printf("Overflow");
    }
    else if (front== -1 && rear== -1)
    {
        front=rear=0;
        queue[rear]=x;
    }
    else
    {
        rear++;
        queue[rear]=x;
    }
}

void dequeue()
{
    if(front== -1 && rear== -1)
    {
        printf("Underflow");
    }
    else if (front==rear)
    {
        front=rear= -1;
    }
    else
    {
        printf("%d", queue[front]);
        front++;
    }
}
```

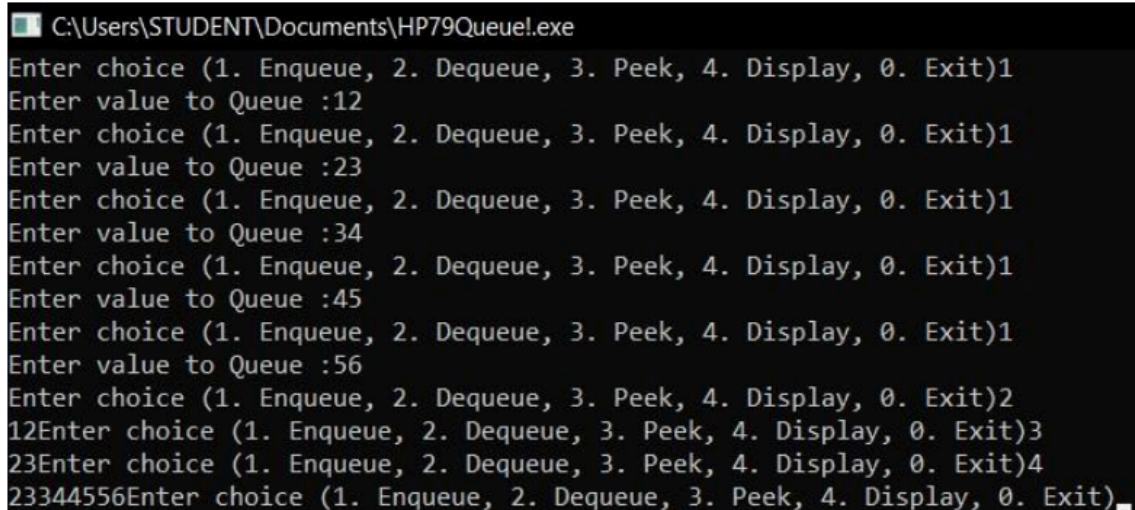


```

int main() {
int ch,value;
do {
printf("Enter the choice: 1:Enqueue,
2:Dequeue, 3:Peek, 4:Display, 0:Exit");
scanf("%d",ch); switch(ch)
{
case1:
printf("Enter the value of enqueue: ");
scanf("%d",value);
enqueue(value);
break;
case2:
dequeue();
break; case3:
peek();
break; case4:
display();
break; case 0:
printf("Exiting.....\n");
break;
}
} while
(ch!=0);
return 0;
}

```

## Output:



```

C:\Users\STUDENT\Documents\HP79Queue!.exe
Enter choice (1. Enqueue, 2. Dequeue, 3. Peek, 4. Display, 0. Exit)1
Enter value to Queue :12
Enter choice (1. Enqueue, 2. Dequeue, 3. Peek, 4. Display, 0. Exit)1
Enter value to Queue :23
Enter choice (1. Enqueue, 2. Dequeue, 3. Peek, 4. Display, 0. Exit)1
Enter value to Queue :34
Enter choice (1. Enqueue, 2. Dequeue, 3. Peek, 4. Display, 0. Exit)1
Enter value to Queue :45
Enter choice (1. Enqueue, 2. Dequeue, 3. Peek, 4. Display, 0. Exit)1
Enter value to Queue :56
Enter choice (1. Enqueue, 2. Dequeue, 3. Peek, 4. Display, 0. Exit)2
12Enter choice (1. Enqueue, 2. Dequeue, 3. Peek, 4. Display, 0. Exit)3
23Enter choice (1. Enqueue, 2. Dequeue, 3. Peek, 4. Display, 0. Exit)4
23344556Enter choice (1. Enqueue, 2. Dequeue, 3. Peek, 4. Display, 0. Exit)0

```

**Conclusion:** This program effectively demonstrates queue operations using an array-based implementation. It handles the primary queue operations like enqueueing and dequeuing while checking for overflow and underflow conditions. The use of a menu-driven interface allows the user to interact with the queue, displaying its contents and checking the front element as needed.

# Experiment No. 5

**Aim:** To implement a circular queue using array ADT.

## Theory:

This program implements a circular queue using an array of fixed size  $N=5$ . A circular queue is a type of queue where the last position is connected back to the first position to form a circle. This allows better use of the available space, as opposed to a linear queue, where elements must be shifted after a dequeue operation.

The key elements of the circular queue are:

Front: The index from which elements are dequeued (removed).

Rear: The index at which elements are enqueued (added).

Queue: The array that holds the queue elements.

Modulo Arithmetic: This ensures that the circular nature of the queue is maintained, by wrapping the array indices using the % operator.

Operations in the Circular Queue:

Enqueue (Insertion):

Adds an element at the rear of the queue.

If the queue is full ( $(\text{rear} + 1) \% N == \text{front}$ ), it prints "Overflow".

If the queue is empty ( $\text{front} == -1 \ \&\& \ \text{rear} == -1$ ), it sets both front and rear to 0.

Otherwise, it increments the rear using modulo arithmetic and inserts the element.

Dequeue (Removal):

Removes an element from the front of the queue.

If the queue is empty ( $\text{front} == -1 \ \&\& \ \text{rear} == -1$ ), it prints "Underflow".

If there's only one element ( $\text{front} == \text{rear}$ ), it resets both front and rear to -1, indicating the queue is empty.

Otherwise, it increments the front using modulo arithmetic to move to the next element.

Peek:

Returns the element at the front without removing it.

If the queue is empty, it prints "Queue is empty."

Display:

Displays all elements in the queue from front to rear, iterating with modulo arithmetic to handle the circular nature.

This structure allows efficient queue management without shifting elements after each dequeue operation, as required in a linear array-based queue.

## Program:

```
#include<stdio.h>
#define N 5
int queue[N];
int front = -1;
int rear = -1;
void display(){
int i = front;
if(front == -1 && rear == -1){
printf("queue is empty");
} else {
printf("Queue is : ");
while(i!=rear){
printf("%d\n", queue[i]);
i = (i+1)%N;
}
printf("%d\n", queue[rear]);
}
}
void enqueue(int x){
if((rear + 1)%N == front){
printf("Overflow");
} else if (front == -1 && rear == -1){
front = rear = 0;
queue[rear] = x;
} else {
rear = (rear + 1)%N;
queue[rear] = x;
}
}
void dequeue(){
if(front == -1 && rear == -1){
printf("Underflow");
} else if (front == rear){
front = -1;
rear = -1;
} else {
printf("%d", queue[front]);
front = (front+1)%N;
}
}
void peek(){
if(front == -1 && rear == -1){
printf("Queue is empty.");
} else {
printf("%d", queue[front]);
}
}
int main(){
int choice, value;
do{
printf("Enter choice : (1. Enqueue 2. Dequeue 3. Peek 4. Display 0. Exit\n");
scanf("%d", &choice);
switch(choice){
case 1: printf("Enter value to enqueue : ");
scanf("%d", &value);
enqueue(value);
break;
case 2: dequeue();
break;
case 3: peek();
break;
case 4 : display();
break;
default : printf("Invalid choice \n");
break;
}
} while(choice!=0);
return 0;
}
```

## Output:

```
Enter choice : (1. Enqueue 2. Dequeue 3. Peek 4. Display 0. Exit
)1
Enter value to enqueue : 69
Enter choice : (1. Enqueue 2. Dequeue 3. Peek 4. Display 0. Exit
)1
Enter value to enqueue : 69
Enter choice : (1. Enqueue 2. Dequeue 3. Peek 4. Display 0. Exit
)2
69Enter choice : (1. Enqueue 2. Dequeue 3. Peek 4. Display 0. Exit
)1
Enter value to enqueue : 123
Enter choice : (1. Enqueue 2. Dequeue 3. Peek 4. Display 0. Exit
)1
Enter value to enqueue : 456
Enter choice : (1. Enqueue 2. Dequeue 3. Peek 4. Display 0. Exit
)3
69Enter choice : (1. Enqueue 2. Dequeue 3. Peek 4. Display 0. Exit
)4
Queue is : 69
123
456
Enter choice : (1. Enqueue 2. Dequeue 3. Peek 4. Display 0. Exit
)
```

## Conclusion:

The program successfully implements a **circular queue** using an array with fixed size  $N=5$ . It demonstrates basic queue operations such as **enqueue**, **dequeue**, **peek**, and **display** with efficient use of the available space. The use of **modulo arithmetic** allows the queue to wrap around, maintaining its circular nature and avoiding the need for element shifting after dequeue operations.

# Experiment No. 6

**Aim:** To Implement double ended using array ADT.

**Theory:** This program implements a Double-Ended Queue (Deque) using a circular array. A Deque allows insertion and deletion of elements from both the front and rear ends, offering more flexibility compared to a regular queue. The program uses a fixed-size array (`deque[size]`), where `size` is defined as 5.

Key Operations of the Deque:

1. Insert at Front (`insert_front`): This function inserts an element at the front of the deque, handling circular behavior using modulo arithmetic to wrap around when necessary.
2. Insert at Rear (`insert_rear`): This function inserts an element at the rear end of the deque and also handles the circular nature of the array.
3. Delete from Front (`delete_front`): This function removes an element from the front of the deque and adjusts the front pointer accordingly.
4. Delete from Rear (`delete_rear`): Similar to `delete_front`, but removes an element from the rear of the deque.
5. Display: This function prints all elements in the deque in order from front to rear, respecting the circular nature of the data structure.
6. Get Front and Rear: These functions return the value at the front and rear of the deque, respectively, without removing them.

The circular nature is managed by updating the `front` and `rear` pointers using modulo arithmetic, ensuring that they stay within the bounds of the array and wrap around when necessary.

## Program:

```
#include<stdio.h>
#define size 5
int deque[size];
int front=-1;
int rear=-1;
void insert_front(int x)
{
    if((front==0 && rear==size-1) ||
(front==rear+1))
    {
        printf("Overflow");
    }
    else if((front== -1) && (rear== -1))
    {
        front=rear=0;
        deque[front]=x;
    }
    else if(front==0)
    {
        front=size-1;
        deque[front]=x;
    }
    else
    {

```

```

front=front-1;
deque[front]=x;
}
}
void insert_rear(int x)
{
if((front==0 &&
rear==size1)||((front==rear+1))
{
printf("Overflow ");
}
else if((front== -1)&&(rear== -1))
{
rear=0;
deque[rear]=x;
}
else if(rear==size-1)
{
rear=0;
deque[rear]=x;
}
else
{
rear++;
deque[rear]=x;
}
}

void display()
{
int i=front;
printf("\nElements in a deque are: ");
while(i!=rear)
{
printf("%d ",deque[i]);
i=(i+1)%size;
}
printf("%d",deque[rear]);
}
void getfront()

```

```

{
if((front== -1) && (rear== -1))
{
printf("Deque is empty");
}
else
{
printf("\nThe value of the element at
front is: %d", deque[front]);
}
}
void getrear()
{
if((front== -1) && (rear== -1))
{
printf("Deque is empty");
}
else
{
printf("\nThe value of the element at rear
is %d", deque[rear]);
}
}
void delete_front()
{
if((front== -1) && (rear== -1))
{
printf("Deque is empty");
}
else if(front==rear)
{
printf("\nThe deleted element is %d",
deque[front]);
front=-1;
rear=-1;
}
else if(front==(size-1))
{
printf("\nThe deleted element is %d",
deque[front]);
}
}

```

```

front=0;
}
else
{
printf("\nThe deleted element is %d",
deque[front]);
front=front+1;
}
}
void delete_rear()
{
if((front==-1) && (rear==-1))
{
printf("Deque is empty");
}
else if(front==rear)
{
printf("\nThe deleted element is %d",
deque[rear]);
front=-1;
rear=-1;
}
else if(rear==0)
{
printf("\nThe deleted element is %d",
deque[rear]);
rear=rear-1;
}
}
int main()
{
insert_front(20);
insert_front(10);
insert_rear(30);
insert_rear(50);
insert_rear(80);
display();
getfront();
getrear();
delete_front();
delete_rear();
display();
return 0;
}

```

## Output:

```

Elements in a deque are: 10 20 30 50 80
The value of the element at front is: 10
The value of the element at rear is 80
The deleted element is 10
The deleted element is 80
Elements in a deque are: 20 30 50
-----
Process exited after 1.752 seconds with return value 0
Press any key to continue . . .

```

**Conclusion:** The program provides a robust implementation of a circular Double-Ended Queue (Deque) using an array. It efficiently handles the addition and removal of elements from both ends, making it more flexible than a standard queue.



# Experiment No. 7

**Aim:** To implement singly linked list

**Theory:** This program implements a Singly Linked List in C, which is a dynamic data structure that allows for efficient insertion, deletion, and traversal of data. Unlike arrays, linked lists do not require contiguous memory allocation, allowing for efficient memory usage, especially when the size of data is not known in advance.

1. Node Structure: The program defines a `struct node` which contains an integer `data` field and a pointer to the next node (`next`).
2. Global Variables:
  - `head`: Pointer to the first node in the linked list.
  - `newnode`: Temporary pointer for creating new nodes.
  - `temp`: Temporary pointer for traversing the list.
3. Functions:
  - `createNode`: Allocates memory for a new node, reads data from the user, and inserts it at the end of the list.
  - `display`: Traverses the linked list from `head` to the end, printing each node's data.
  - `insertAtBeginning`: Inserts a new node at the beginning of the list.
  - `insertAtEnd`: Inserts a new node at the end of the list.
  - `getLength`: Returns the total number of nodes in the list.
  - `insertAfterGivenPosition`: Inserts a new node after a specified position in the list.
  - `DeleteFromBeginning`: Deletes the first node from the list.
  - `DeleteFromEnd`: Deletes the last node from the list.
  - `DeleteFromPosition`: Deletes a node at a specified position.
  - `reverse`: Reverses the linked list in place.
4. Main Function: Implements a menu-driven interface allowing users to choose different operations on the linked list, such as creating nodes, displaying the list, inserting nodes, deleting nodes, and reversing the list.

## Program:

```
#include <stdio.h>                                int data;

#include <stdlib.h>                                struct node *next;

struct node                                       };

{
```

```
struct node *head, *newnode, *temp;
```

```
void createnode()
```

```
{
```

```
    newnode = (struct  
node*)malloc(sizeof(struct node));
```

```
    printf("Enter Data ");
```

```
    scanf("%d", &newnode->data);
```

```
    newnode->next=0;
```

```
    if(head==0){
```

```
        head=temp=newnode;
```

```
    }
```

```
    else{
```

```
        temp->next=newnode;
```

```
        temp = newnode;
```

```
    }
```

```
}
```

```
void display()
```

```
{
```

```
    temp=head;
```

```
    while(temp!=0){
```

```
        printf("%d ", temp->data);
```

```
        temp=temp->next;
```

```
    }
```

```
    printf("\n");
```

```
}
```

```
void insertAtBeginning()
```

```
{
```

```
    newnode = (struct node  
*)malloc(sizeof(struct node));
```

```
    printf("Enter Data at Beginning ");
```

```
    scanf("%d", &newnode->data);
```

```
    newnode->next = head;
```

```
    head = newnode;
```

```
}
```

```
void insertAtEnd()
```

```
{
```

```
    newnode = (struct node  
*)malloc(sizeof(struct node));
```

```
    printf("Insert data at END ");
```

```
    scanf("%d", &newnode->data);
```

```
    newnode->next=0;
```

```
    temp = head;
```

```
    while(temp->next!=0)
```

```

        {
            temp = temp->next;
        }
        temp->next=newnode;
    }

```

```
int getLength()
```

```

{
    temp=head;
    int count=0;

    while(temp!=0){
        count++;
        temp=temp->next;
    }
    return count;
}

```

```
void insertAfterGivenPosition()
```

```

{
    int position, i=1;
    printf("Enter position: ");
    scanf("%d", &position);

```

```

    if(position>getLength()){

```

```

        printf("INVALID POSITION");
    }

```

```
    else{
```

```
        temp=head;
```

```
        while(i<position-1){
```

```
            temp=temp->next;
```

```
            i++;
```

```
        }
```

```
    }
```

```

        newnode = (struct node
*)malloc(sizeof(struct node));

```

```
        printf("Enter Data: ");
```

```
        scanf("%d", &newnode->data);
```

```
        newnode->next = temp->next;
```

```
        temp->next = newnode;
```

```
    }
```

```
void DeleteFromBeginning()
```

```
{
```

```
    struct node *temp;
```

```
    temp=head;
```

```
    head= head->next;
```

```
    free(temp);
```

```
}
```

```
void DeleteFromEnd()
```

```

{
    struct node *prevnode;

    temp=head;

    while(temp->next!=0)
    {
        prevnode=temp;
        temp=temp->next;
    }
    if(temp==head)
    {
        head=0;
    }
    else{
        prevnode->next=0;
    }
    free(temp);
}

```

```

void DeleteFromPosition()
{
    struct node *nextnode;

    int position, i=1;

    printf("Enter Position ");

```

```

scanf("%d", &position);

    if(position>getLength())
    {
        printf("INVALID POSITION!!!!\n");
    }
    else if(position==1){
        DeleteFromBeginning();
    }
    else if(position==getLength()){
        DeleteFromEnd();
    }
    else{
        temp=head;
        while(i<position-1)
        {
            temp=temp->next;
            i++;
        }
        nextnode = temp->next;
        temp->next = nextnode->next;
        free(nextnode);
    }
}

```

```

void reverse()
{
    struct node *prevnode, *currentnode,
    *nextnode;

    prevnode=0;
    currentnode=nextnode=head;

    while(nextnode!=0)
    {
        nextnode=nextnode->next;
        currentnode->next=prevnode;
        prevnode=currentnode;
        currentnode=nextnode;
    }
    head=prevnode;
}

```

```

int main()
{
    int ch=1;

    do
    {
        printf("Enter choice-\n 1- Create
Node\n 2- Display\n 3- Insert At
Beginning\n 4- Insert At End\n 5- Insert
After Given Position\n 6-Delete From

```

Beginning\n 7-Delete From End\n 8-Delete  
From Specific Position\n 9-Reverse\n");

```

scanf("%d", &ch);

switch(ch)
{
    case 1: createnode();
        break;
    case 2: display();
        break;
    case 3: insertAtBeginning();
        break;
    case 4: insertAtEnd();
        break;
    case 5: insertAfterGivenPosition();
        break;
    case 6: DeleteFromBeginning();
        break;
    case 7: DeleteFromEnd();
        break;
    case 8: DeleteFromPosition();
        break;
    case 9: reverse();
        break;
    default: printf("INVALID CHOICE!!!");
}

```

```

    }
while(ch!=0);
    }

```

## Output:

```

C:\Users\STUDENT.LAB 9 LAPTOP 27\Desktop\javany:
Enter choice-
1- Create Node
2- Display
3- Insert At Beginning
4- Insert At End
5- Insert After Given Position
6-Delete From Beginning
7-Delete From End
8-Delete From Specific Position
9-Reverse
Enter Data 2
Enter choice-
1- Create Node
2- Display
3- Insert At Beginning
4- Insert At End
5- Insert After Given Position
6-Delete From Beginning
7-Delete From End
8-Delete From Specific Position
9 Reverse

```

```

Enter Data 5
Enter choice-
1- Create Node
2- Display
3- Insert At Beginning
4- Insert At End
5- Insert After Given Position
6-Delete From Beginning
7-Delete From End
8-Delete From Specific Position
9-Reverse
Enter Data 9
Enter choice-
1- Create Node
2- Display
3- Insert At Beginning
4- Insert At End
5- Insert After Given Position
6-Delete From Beginning
7-Delete From End
8-Delete From Specific Position
9-Reverse
Enter Data 3
Enter choice
1- Create Node

```

C:\Users\STUDENT.LAB-9-LAPTOP-27\Desktop\javanya 87.exe

```
Enter choice-
1- Create Node
2- Display
3- Insert At Beginning
4- Insert At End
5- Insert After Given Position
6-Delete From Beginning
7-Delete From End
8-Delete From Specific Position
9-Reverse
2
2 5 9 3
Enter choice-
1- Create Node
2- Display
3- Insert At Beginning
4- Insert At End
5- Insert After Given Position
6-Delete From Beginning
7-Delete From End
8-Delete From Specific Position
9-Reverse
```

```
3
Enter Data at Beginning 9
Enter choice-
1- Create Node
2- Display
3- Insert At Beginning
4- Insert At End
5- Insert After Given Position
6-Delete From Beginning
7-Delete From End
8-Delete From Specific Position
9-Reverse
2
9 2 5 9 3
Enter choice-
1- Create Node
2- Display
3- Insert At Beginning
4- Insert At End
5- Insert After Given Position
6-Delete From Beginning
7-Delete From End
8-Delete From Specific Position
9-Reverse
6
Enter choice-
1- Create Node
2- Display
```

**Conclusion:** The program successfully implements a basic Singly Linked List with various operations, demonstrating the flexibility and dynamic nature of linked lists. The ability to insert and delete nodes at any position, including at the beginning and end, showcases the advantages of linked lists over static data structures like arrays.

# Experiment No. 8

**Aim:** To implement stack using Linked List

**Theory:** This program implements a Stack using a Linked List in C. A stack is a linear data structure that follows the Last In, First Out (LIFO) principle, meaning the last element added to the stack will be the first one removed. A stack can be implemented using either an array or a linked list, and in this program, we use a dynamic approach through linked lists to manage stack operations.

**Key Concepts:**

1. Node Structure: The program defines a **struct node** that contains two fields:
  - **data**: Holds the integer value of the stack element.
  - **link**: A pointer to the next node in the linked list.
2. Global Variable:
  - **top**: A pointer that tracks the top element of the stack.
3. Operations Implemented:
  - **push(x)**: Adds (or "pushes") an element **x** to the top of the stack.
  - **pop()**: Removes (or "pops") the top element from the stack and displays it. It frees the memory of the popped node.
  - **peek()**: Displays the value of the top element without removing it from the stack.
  - **display()**: Traverses and displays all elements in the stack from top to bottom.

**Working:**

1. Push Operation:
  - A new node is created, its **data** is set to the given value, and it is linked to the current top node.
  - The new node becomes the new top of the stack.
2. Pop Operation:
  - If the stack is empty, an underflow message is printed.
  - If the stack has elements, the top element is removed, and the **top** pointer is updated to the next node in the stack. The removed node is freed to prevent memory leaks.
3. Peek Operation:
  - It checks if the stack is empty. If not, it displays the top element without modifying the stack.
4. Display Operation:
  - It traverses the stack from the **top** to the bottom, printing each element.



The main function provides a menu-driven interface allowing the user to perform various operations on the stack, such as pushing, popping, peeking, and displaying stack elements.

### Program:

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

struct node
{
    int data;
    struct node *link;
};

struct node *top=0;

void push(int x)
{
    struct node *new_node;
    new_node=(struct
node*)malloc(sizeof(struct node));
    new_node->data=x;
    new_node->link=top;
    top=new_node;
}

void pop()
{
    struct node *temp;
    temp=top;
    if(top==0)
    {
        printf("STACK is empty");
    }
    else
    {
        printf("%d",top->data);
    }
}

void peek()
{
    if(top==0)
    {
        printf("STACK is Empty");
    }
    else
    {
        printf("The top most element in the
Stack is: %d",top->data);
    }
}

void display()
{
    struct node *temp;
    temp=top;
    if(top==0)
    {
        printf("Stack is empty");
    }
    else
    {
        while(temp!=0)
        {
            printf("%d",temp->data);
            temp=temp->link;
            free(temp);
        }
    }
}
```

```

        {
            printf("%d",temp->data);
            temp=temp->link;
        }
    }
}

int main()
{
    struct node *top=0;
    int ch,x;
    do
    {
        printf("\nEnter choise:\n1:push, 2:pop(),
3:peek(), 4:display()");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
                printf("Enter value to push:");
                scanf("%d", &x);
                push(x);
                break;
            case 2:
                pop();
                break;
            case 3:
                peek();
                break;
            case 4:
                display();
                break;

            default:
                printf("\nInvalid Input\n");
        }
    }while(ch!=0);
}

```

## Output:

```

Enter choise:
1:push, 2:pop(), 3:peek(), 4:display():1
Enter value to push:22

Enter choise:
1:push, 2:pop(), 3:peek(), 4:display():1
Enter value to push:4

Enter choise:
1:push, 2:pop(), 3:peek(), 4:display():4
422
Enter choise:
1:push, 2:pop(), 3:peek(), 4:display():3
The top most element in the Stack is: 4
Enter choise:
1:push, 2:pop(), 3:peek(), 4:display():2
4
Enter choise:
1:push, 2:pop(), 3:peek(), 4:display():_

```

**Conclusion:** The program successfully implements a Stack using a Linked List, providing dynamic memory management and allowing the stack to grow and shrink as needed without worrying about predefining its size (as would be the case in an array-based implementation). The linked list structure allows efficient insertion and deletion of elements at the top of the stack in constant time.

# Experiment No. 9

**Aim:** To implement Binary Search Tree ADT using Linked list.

**Theory:** The program implements a binary search tree (BST) in C, which is a hierarchical data structure that allows efficient searching, insertion, and deletion operations.

Key Components of the Program:

## 1. Data Structure:

- A **struct BstNode** is defined to represent each node in the binary search tree, containing three members:
  - **data**: an integer value stored in the node.
  - **left**: a pointer to the left child node.
  - **right**: a pointer to the right child node.

## 2. Insertion Function:

- The **insert** function takes the root of the BST and an integer value as parameters. It creates a new node and inserts it into the correct position in the tree:
  - If the tree is empty (**root** is **NULL**), the new node becomes the root.
  - If the tree is not empty, it compares the new value with the current node's value and decides whether to traverse left or right recursively until it finds the correct position.

## 3. Search Function:

- The **search** function checks if a particular value exists in the BST:
  - If the **root** is **NULL**, the value is not found.
  - If the value matches the current node's value, it returns **1** (found).
  - If the value is less than the current node's value, it searches the left subtree; otherwise, it searches the right subtree.

## 4. Main Function:

- The program inserts several integers into the BST (15, 10, 20, 5, 30, 25).
- It enters an infinite loop that prompts the user to search for values in the BST. The loop continues until the user chooses to end it.

## Program:

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

struct BstNode
{
    int data;
    struct BstNode* left;
```

```

    struct BstNode* right;

};

struct BstNode* root=NULL;

struct BstNode* insert(struct BstNode*
root,int x)
{
    struct BstNode* temp=(struct
BstNode*)malloc(sizeof(struct BstNode));

    temp->data=x;

    temp->left=temp->right=NULL;

    if(root==NULL)

        root=temp;

    else if(root->data>=x)

        root->left=insert(root->left,x);

    else

        root->right=insert(root->right,x);

    return root;

}

int search(struct BstNode* root,int data)

{

    if(root==NULL)

```

```

        return 0;

    else if(root->data == data) {

        return 1;}

    else if(root->data>=data)

        return search(root->left,data);

    else

        return search(root->right,data);

}

int main()

{

    root = insert(root, 15);

    root = insert(root, 10);

    root = insert(root, 20);

    root = insert(root, 05);

    root = insert(root, 30);

    root = insert(root, 25);

    while(1)

    {

```

int n,l,k;	if(search(root,n)==1)
printf("press 1 for continue and press 2 for end ");	printf("found in\n");
	else
scanf("%d",&k);	printf("not found in\n");
if(k==1)	}
{	else
printf("enter the value for searching \n");	break;
scanf("%d",&n);	}
	}

### Output:

```

press 1 for continue and press 2 for end 1
enter the value for searching
12
not found in
press 1 for continue and press 2 for end 1
enter the value for searching
20
found in
press 1 for continue and press 2 for end _

```

**Conclusion:** The program effectively demonstrates the basic operations of a binary search tree, specifically insertion and search.

# Experiment No. 10.1

**Aim:** Implement Graph traversal techniques : a) Depth First Search

**Theory:** The program implements the Breadth-First Search (BFS) algorithm to traverse a graph. BFS is an algorithm used for searching a tree or graph data structure. It starts at a selected node (often called the "root" in trees) and explores all of its neighboring nodes at the present depth before moving on to nodes at the next depth level.

1. Data Structures:

- Graph Representation: The graph is represented using an adjacency list, where each vertex has a linked list of its adjacent vertices. This is implemented using a `struct node` for the vertices and a `struct Graph` that contains an array of these nodes and a visited array to track visited vertices.
- Queue for BFS: A queue is implemented using a `struct queue` to facilitate the BFS traversal. The queue stores the vertices to be explored, ensuring that nodes are visited in the correct order.

2. BFS Algorithm:

- The `bfs` function starts at a specified vertex, marking it as visited and enqueueing it. It repeatedly dequeues a vertex, visits it, and enqueues its unvisited neighbors until all reachable vertices are visited.

3. Graph Initialization and Edge Addition:

- The graph is initialized with a specified number of vertices. The `addEdge` function adds edges between vertices to create the graph structure.

4. Main Function:

- The `main` function initializes a graph with 6 vertices, adds edges to define its structure, and then calls the BFS function starting from vertex 0.

## Program:

```
// BFS algorithm in C
#include <stdio.h>
#include <stdlib.h>
#define SIZE 40
struct queue {
    int items[SIZE];
    int front;
    int rear;
};
struct queue* createQueue();

void enqueue(struct queue* q, int);
int dequeue(struct queue* q);
void display(struct queue* q);
int isEmpty(struct queue* q);
void printQueue(struct queue* q);

struct node {
    int vertex;
    struct node* next;
};
```

```

struct node* createNode(int);

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

// BFS algorithm
void bfs(struct Graph* graph, int
startVertex) {
    struct queue* q = createQueue();
    graph->visited[startVertex] = 1;
    enqueue(q, startVertex);
    while (!isEmpty(q)) {
        printQueue(q);
        int currentVertex = dequeue(q);
        printf("Visited %d\n", currentVertex);
        struct node* temp =
graph->adjLists[currentVertex];
        while (temp) {
            int adjVertex = temp->vertex;
            if (graph->visited[adjVertex] == 0) {
                graph->visited[adjVertex] = 1;
                enqueue(q, adjVertex);
            }
            temp = temp->next;
        }
    }
}

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

// Creating a graph

```

```

struct Graph* createGraph(int vertices) {
    struct Graph* graph = malloc(sizeof(struct
Graph));
    graph->numVertices = vertices;
    graph->adjLists = malloc(vertices *
sizeof(struct node*));
    graph->visited = malloc(vertices *
sizeof(int));
    int i;
    for (i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }
    return graph;
}

// Add edge
void addEdge(struct Graph* graph, int src,
int dest) {
    // Add edge from src to dest
    struct node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;
    // Add edge from dest to src
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// Create a queue
struct queue* createQueue() {
    struct queue* q = malloc(sizeof(struct
queue));
    q->front = -1;
    q->rear = -1;
    return q;
}

// Check if the queue is empty
int isEmpty(struct queue* q) {
    if (q->rear == -1)

```



```

        return 1;
    else
        return 0;
}

// Adding elements into queue
void enqueue(struct queue* q, int value) {
    if (q->rear == SIZE - 1)
        printf("\nQueue is Full!!");
    else {
        if (q->front == -1)
            q->front = 0;
        q->rear++;
        q->items[q->rear] = value;
    }
}

```

```

// Removing elements from queue
int dequeue(struct queue* q) {
    int item;
    if (isEmpty(q)) {
        printf("Queue is empty");
        item = -1;
    } else {
        item = q->items[q->front];
        q->front++;
        if (q->front > q->rear) {
            printf("Resetting queue ");
            q->front = q->rear = -1;
        }
    }
}

```

```

    }
    return item;
}

```

```

// Print the queue
void printQueue(struct queue* q) {
    int i = q->front;
    if (isEmpty(q)) {
        printf("Queue is empty");
    } else {
        printf("\nQueue contains \n");
        for (i = q->front; i < q->rear + 1; i++) {
            printf("%d ", q->items[i]);
        }
    }
}

```

```

int main() {
    struct Graph* graph = createGraph(6);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 4);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 4);

    bfs(graph, 0);
    return 0;
}

```

### Output:

```
Queue contains  
0 Resetting queue Visited 0  
  
Queue contains  
2 1 Visited 2  
  
Queue contains  
1 4 Visited 1  
  
Queue contains  
4 3 Visited 4  
  
Queue contains  
3 Resetting queue Visited 3
```

**Conclusion:** The program successfully demonstrates the BFS algorithm applied to a graph using an adjacency list representation.

## Experiment No. 10.2

**Aim:** Implement Graph traversal techniques : a) BreadthFirst Search

**Theory:** The program implements the Depth-First Search (DFS) algorithm, which is a fundamental algorithm used to traverse or search through graph data structures. The DFS algorithm starts at a selected node (the "source" or "root") and explores as far as possible along each branch before backtracking. This exploration continues until all reachable vertices from the source have been visited.

1. Data Structures:

- Graph Representation: The graph is represented using an adjacency list, where each vertex is associated with a linked list of its adjacent vertices. This is implemented using a `struct node` to represent each vertex and its edges, while the `struct Graph` holds an array of these nodes (adjacency lists) and a visited array to track which vertices have been visited.

2. DFS Algorithm:

- The `DFS` function takes a graph and a starting vertex as input. It marks the vertex as visited, prints it, and recursively visits all its unvisited neighbors. This continues until all vertices reachable from the starting vertex are explored.

3. Graph Initialization and Edge Addition:

- The graph is initialized with a specified number of vertices using the `createGraph` function. Edges between vertices are added using the `addEdge` function, which ensures that the graph remains undirected.

4. Display Function:

- The `displayGraph` function prints the adjacency list representation of the graph, making it easier to visualize the connections between vertices.

5. Main Function:

- The `main` function initializes a graph with 8 vertices, adds edges to define its structure, displays the graph, and then performs a DFS traversal starting from vertex 1.

### Program:

```
// dfs program in C
#include <stdio.h>
#include <stdlib.h>
```

```
struct node {
    int vertex;
    struct node* next;
};
```

```
struct node* createNode(int v);
```

```
struct Graph {
    int totalVertices;
    int* visited;
    struct node** adjLists;
};
```

```

void DFS(struct Graph* graph, int vertex) {
    struct node* adjList =
graph->adjLists[vertex];
    struct node* temp = adjList;

    graph->visited[vertex] = 1;
    printf("%d -> ", vertex);

    while (temp != NULL) {
        int connectedVertex = temp->vertex;

        if (graph->visited[connectedVertex] == 0)
        {
            DFS(graph, connectedVertex);
        }
        temp = temp->next;
    }
}

```

```

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

```

```

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

    graph->adjLists = malloc(vertices *
sizeof(struct node*));
    graph->visited = malloc(vertices *
sizeof(int));
    int i;
    for (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 displayGraph(struct Graph* graph) {
    int v;
    for (v = 1; v < graph->totalVertices; v++) {
        struct node* temp = graph->adjLists[v];
        printf("\n%d => ", v);
        while (temp) {
            printf("%d, ", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }
    printf("\n");
}

```

```

int main() {
    struct Graph* graph = createGraph(8);
    addEdge(graph, 1, 5);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 3, 6);
    addEdge(graph, 2, 7);
    addEdge(graph, 2, 4);
}

```

```
printf("\nThe Adjacency List of the Graph\nDFS(graph, 1);
is:");
displayGraph(graph);
printf("\nDFS traversal of the graph: \n");
return 0;
}
```

**Output:**

```
The Adjacency List of the Graph is:
1 => 3, 2, 5,
2 => 4, 7, 1,
3 => 6, 1,
4 => 2,
5 => 1,
6 => 3,
7 => 2,

DFS traversal of the graph:
1 -> 3 -> 6 -> 2 -> 4 -> 7 -> 5 ->
```

**Conclusion:** The program effectively implements the Depth-First Search (DFS) algorithm, demonstrating its efficiency in traversing graph structures.