Vidyavardhini's College of Engineering and Technology

Department of Artificial Intelligence & Data Science

Course name/Code: Data Structures (CSC303)

List of Experiments

S.No.	Name of Experiment	DOP	DOC	Marks	Sign
Basic Experiments					
1	Implement Stack ADT using array.	19/7/23	26/7/23		
2	Convert an Infix expression to Postfix expression.	26/7/23	2/8/23		
3	Evaluate Postfix Expression.	2/8/23	9/8/23		
4	Implementation of Queue menu driven program using arrays	9/8/23	16/8/23		
5	Implement Circular Queue ADT using array.	16/8/23	23/8/23		
6	Implement Singly Linked List ADT.	23/8/23	6/9/23		
7	Implement Circular Linked List ADT.	6/9/23	13/9/23		
8	Implement Stack ADT using Linked List.	13/9/23	27/9/23		
9	Implementation Binary Search Tree using Linked List	27/9/23	4/10/23		
10	Implementation of Graph traversal techniques - Breadth First Search and Depth First Search	4/10/23	11/10/23		
Assignments					
1					
2					
3					
4					
5					
6					

Course In-charge

(Ms. Neha Raut)

Experiment No.1
Implement Stack ADT using array.
Name: Sing Rahul Rammilan
Roll no:56
Date of Performance: 19/7/23
Date of Submission: 26/7/23
Marks:
Sign:

Experiment No. 1: To implement stack ADT using arrays

Aim: To implement stack ADT using arrays.

Objective:

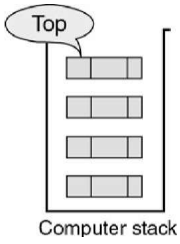
1) Understand the Stack Data Structure and its basic operators.

2) Understand the method of defining stack ADT and implement the basic operators.

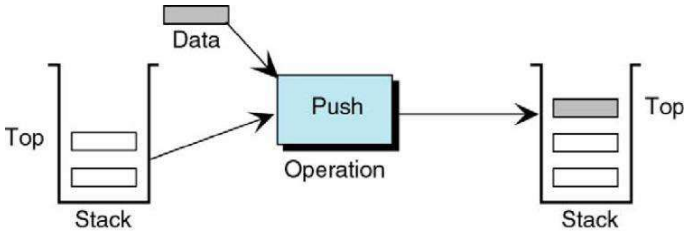
3) Learn how to create objects from an ADT and invoke member functions.

Theory:

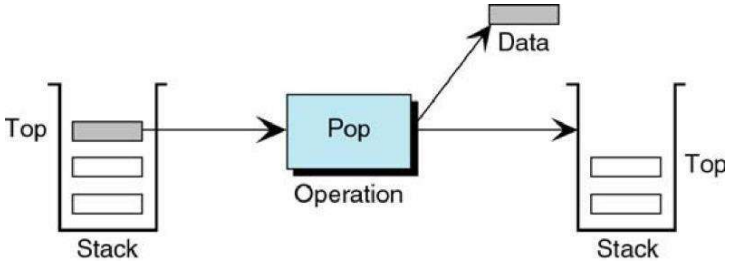
A stack is a data structure where all insertions and deletions occur at one end, known as the top. It follows the Last In First Out (LIFO) principle, meaning the last element added to the stack will be the first to be removed. Key operations for a stack are "push" to add an element to the top, and "pop" to remove the top element. Auxiliary operations include "peek" to view the top element without removing it, "isEmpty" to check if the stack is empty, and "isFull" to determine if the stack is at its maximum capacity. Errors can occur when pushing to a full stack or popping from an empty stack, so "isEmpty" and "isFull" functions are used to check these conditions. The "top" variable is typically initialized to -1 before any insertions into the stack.



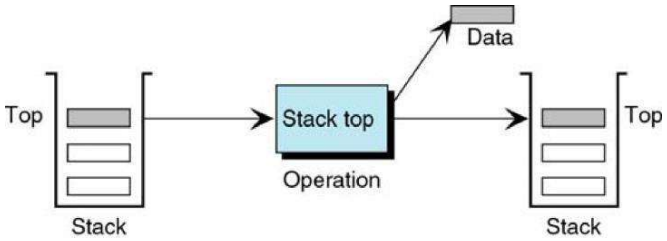
Push Operation



Pop Operation



Peek Operation



Algorithm:

PUSH(item)

1. If (stack is full)

   Print "overflow"

2. top = top + 1
3. stack[top] = item

   Return

POP()

1. If (stack is empty)

   Print "underflow"

2. Item = stack[top]
3. top = top – 1
4. Return item

PEEK()

1. If (stack is empty)

   Print "underflow"

2. Item = stack[top]

3. Return item

ISEMPTY()

1. If(top = -1)then

return 1

2. return 0

ISFULL()

1. If(top = max)then

return 1

2. return 0

Code:

```
#include <stdio.h>
int stack[100],choice,n,top,x,i;
void push(void);
 void pop(void);
 void display(void);
 int main() {
 //clrscr();
 top=-1;
 printf("\n Enter the size of STACK[MAX=100]:");
 scanf("%d",&n);
printf("\n\t STACK OPERATIONS USING ARRAY"); printf("\n\t----------
");
printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
 do { printf("\n Enter the Choice:");
```

```
scanf("%d",&choice);
switch(choice) {
 case 1: {
 push();
 break;
 } case 2: {
 pop();
break;
} case 3: {
 display();
 break;
 } case 4: {
 printf("\n\t EXIT POINT ");
 break;
} default: {
 printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
}
 }
 } while(choice!=4);
 return 0;
} void push() {
if(top>=n-1) {
 printf("\n\tSTACK is over flow"); }
 else {
 printf(" Enter a value to be pushed:");
 scanf("%d",&x);
```

```
top++; stack[top]=x; }
 }
 void pop() { if(top<=-1) {
 printf("\n\t Stack is under flow"); }
 else {
printf("\n\t The popped elements is %d",stack[top]);
 top--;
}
} void display() {
 if(top>=0) {
 printf("\n The elements in STACK \n");
 for(i=top; i>=0; i--)
printf("\n%d",stack[i]);
 printf("\n Press Next Choice"); }
else { printf("\n The STACK is empty"); } }
```

Output:

```
Enter the size of STACK[MAX=100]:10
STACK OPERATIONS USING ARRAY
    --------------------------------
     1.PUSH
     2.POP
     3.DISPLAY
     4.EXIT
 Enter the Choice:1
 Enter a value to be pushed:2
 Enter the Choice:1
 Enter a value to be pushed:5
 Enter the Choice:2
 The popped elements is 5
 Enter the Choice:1
 Enter a value to be pushed:4
 Enter the Choice:3
 The elements in STACK

4
2
```

Conclusion:

What is the structure of Stack ADT?

The Stack Abstract Data Type (ADT) is a linear data structure that follows the Last-In-First-Out (LIFO) principle. It has two primary operations:

1. Push: This operation adds an element to the top of the stack.

2. Pop: This operation removes and returns the element from the top of the stack.

In addition to these fundamental operations, a stack typically has the following characteristics:

- Peek (or Top): This allows you to view the element at the top of the stack without removing it.

- IsEmpty: This function checks if the stack is empty.

- Size: This function returns the number of elements in the stack.

A simple real-life analogy is a stack of plates: you add plates to the top (Push) and remove them from the top (Pop).

List various applications of stack?

1. Expression evaluation

2. Function call management

3. Backtracking algorithms

4. Undo functionality

5. Memory management

6. Syntax parsing

7. Browser history

8. Task management

9. Expression matching

10. Postfix calculations

11. Undo/redo in text editors

12. Playlists

13. Call history

14. Task scheduling

15. Routing algorithms

16. Symbol balancing

17. Navigation systems

Which stack operation will be used when the recursive function call is returning to the calling function?

The stack operation used when a recursive function call is returning to the calling function is called "Pop." When the recursive function completes its execution and returns a value, the information related to that function call (such as local variables, return address, and other context) is popped or removed from the call stack. This allows the program to resume execution in the calling function from the point where the recursive call was made.

Experiment No.2
Convert an Infix expression to Postfix expression using stack ADT.
Name: Singh Rahul Rammilan
Roll No: 56
Date of Performance:26/7/23
Date of Submission:2/8/23
Marks:
Sign:

.

Experiment No. 2: Conversion of Infix to postfix expression using stack ADT

Aim: To convert infix expression to postfix expression using stack ADT.

Objective:

1) Understand the use of Stack.

2) Understand how to import an ADT in an application program.

3) Understand the instantiation of Stack ADT in an application program.

4) Understand how the member functions of an ADT are accessed in an application program.

Theory:

Postfix notation is a way of representing algebraic expressions without parentheses or operator precedence rules. In this notation, expressions are evaluated by scanning them from left to right and using a stack to perform the calculations. When an operand is encountered, it is pushed

onto the stack, and when an operator is encountered, the last two operands from the stack are popped and used in the operation, with the result then pushed back onto the stack. This process continues until the entire postfix expression is parsed, and the result remains in the stack.

Conversion of infix to postfix expression

Expression	Stack	Output
2	Empty	2
*	*	2
3	*	23
/	/	23*
(	/(	23*
2	/(	23*2
-	/(-	23*2
1	/(-	23*21
)	/	23*21-
+	+	23*21-/
5	+	23*21-/5
*	+*	23*21-/53
3	+*	23*21-/53
	Empty	23*21-/53*+

Algorithm:

Conversion of infix to postfix

Step 1: Add ")" to the end of the infix expression
Step 2: Push "(" on to the stack
Step 3: Repeat until each character in the infix notation is scanned
IF a "(" is encountered,push it on the stack
IF an operand (whether a digit or a character) is encountered, add it to the postfix expression.
IF a ")" is encountered, then
a. Repeatedly pop from stack and add it to the postfix expression until a "(" is encountered.
b. Discard the "(". That is, remove the "(" from stack and do not add it to the postfix expression
IF an operator 0 is encountered, then

a. Repeatedly pop from stack and add each operator (popped from the stack) to t postfix expression which has the same precedence or a higher precedence than o
b. Push the operator o to the stack
[END OF IF]

Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty

Step 5: EXIT

Code:

```
#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:

Conclusion:

Convert the following infix expression to postfix (A+(C/D))*B

scanned	stack	pe
(	(	empty
A	(	A
+	+	A
(	+(	A
C	+(	AC
/	+(/	AC
D	+(/	ACD
)	+	ACD/
)		ACD/+
*	*	ACD/+
B	*	ACD/+B
		ACD/+B*

How many push and pop operations were required for the above conversion?

In the given conversion of the infix expression "A+(C/D)*B" to postfix notation

Push Operations: 9

Pop Operations: 9

There were a total of 9 push operations and 9 pop operations during the conversion from infix to postfix notation.

Where is the infix to postfix conversion used or applied?

Infix to postfix conversion is applied in:

1. Calculator software.
2. Programming language compilers.
3. Expression evaluation.
4. Mathematical software.
5. Spreadsheet programs.
6. Computer algebra systems.
7. Calculator hardware.
8. Query languages.
9. Expression parsing.
10. Scientific and engineering simulations.

Experiment No.3
Evaluate Postfix Expression using Stack ADT.
Name: Singh Rahul Rammilan
Roll No: 56
Date of Performance: 2/8/23
Date of Submission: 9/8/23
Marks:
Sign:

Experiment No. 3: Evaluation of Postfix Expression using stack ADT

Aim : Implementation of Evaluation of Postfix Expression using stack ADT

Objective:

1) Understand the use of Stack.
2) Understand importing an ADT in an application program.
3) Understand the instantiation of Stack ADT in an application program.
4) Understand how the member functions of an ADT are accessed in an application program

Theory:

An arithmetic expression consists of operands and operators. For a given expression in a postfix form, stack can be used to evaluate the expression. The rule is whenever an operand comes into the string, push it onto the stack and when an operator is found then the last two elements from the stack are popped and computed and the result is pushed back onto the stack. One by one the whole string of postfix expressions is parsed and the final result is obtained at an end of computation that remains in the stack.

Algorithm

Step 1: Add a ")" at the end of the postfix expression
Step 2: Scan every character of the postfix expression and repeat Steps 3 and 4 until ")"is encountered
Step 3: IF an operand is encountered, push it on the stack
IF an operator 0 is encountered, then
a. Pop the top two elements from the stack as A and B as A and B
b. Evaluate BOA, where A is the topmost element and B is the element below A.
c. Push the result of evaluation on the stack [END OF IF]
Step 4: SET RESULT equal to the topmost element of the stack
Step 5: EXIT

Code:

```
#include<stdio.h>

#include<ctype.h>

 #define MAXSTACK 100

#define POSTFIXSIZE 100

 int stack[MAXSTACK];

int top = -1;

 void push(int item) {

if (top >= MAXSTACK - 1) {

 printf("stack over flow");

 return;

} else {

 top = top + 1;
```

```
 stack[top] = item; }
}
 int pop() {
 int item;
if (top < 0) {
 printf("stack under flow");
 } else {
 item = stack[top];
 top = top - 1;
 return item;
}
 }
void EvalPostfix(char postfix[]) {
 int i; char ch; int val; int A, B;
for (i = 0; postfix[i] != ')'; i++) {
ch = postfix[i];
 if (isdigit(ch)) {
 push(ch - '0'); }
 else if (ch == '+' || ch == '-' || ch == '*' || ch == '/') {
 A = pop();
 B = pop();
 switch (ch) {
 case '*': val = B * A;
break;
 case '/': val = B / A;
break;
```

```
case '+': val = B + A;
break;
case '-': val = B - A;
break;
} push(val); }
} printf(" \n Result of expression evaluation : %d \n", pop());
} int main() {
int i;
char postfix[POSTFIXSIZE];
printf("ASSUMPTION: There are only four operators(*, /, +, -) in an expression and operand is single digit only.\n");
printf(" \nEnter postfix expression,\npress right parenthesis ')' for end expression : ");
for (i = 0; i <= POSTFIXSIZE - 1; i++) {
scanf("%c", &postfix[i]); if (postfix[i] == ')')
{
break;
}
} EvalPostfix(postfix);
return 0;
}
```

Output:

```
/tmp/Uux1dFb17F.o
ASSUMPTION: There are only four operators(*, /, +, -) in an expression
    and operand is single digit only.

Enter postfix expression,
press right parenthesis ')' for end expression : (1
+56/-46+)stack under flow
 Result of expression evaluation : 10
```

Conclusion:

Elaborate the evaluation of the following postfix expression in your program.

AB+C-

Will this input be accepted by your program. If so, what is the output?

To evaluate the postfix expression "AB+C-," follow these steps:

1. - When you encounter an operand (A), push it onto the stack.

2.- When you encounter another operand (B), push it onto the stack.

3. When you encounter an operator (+), pop the top two operands from the stack (B and A in this case), perform the addition (A + B), and push the result back onto the stack.

4. Continue scanning the expression.

5. When you encounter the next operator (-), pop the top two operands from the stack (result of A + B and C in this case), perform the subtraction (A + B - C), and push the final result back onto the stack.

6. After processing the entire expression, the final result is the only item left in the stack, which is "A + B - C."

So, the output of evaluating the postfix expression "AB+C-" is "A + B - C."

This input would be accepted by a program designed to evaluate postfix expressions, and the output would be "A + B - C."

Experiment No.4
Implementation of Queue menu driven program using arrays
Name: Singh Rahul Rammilan
Roll No: 56
Date of Performance: 9/8/23
Date of Submission: 16/8/23
Marks:
Sign:

Experiment No. 4: Simple Queue Operations

Aim: To implement a Linear Queue using arrays.

Objective:

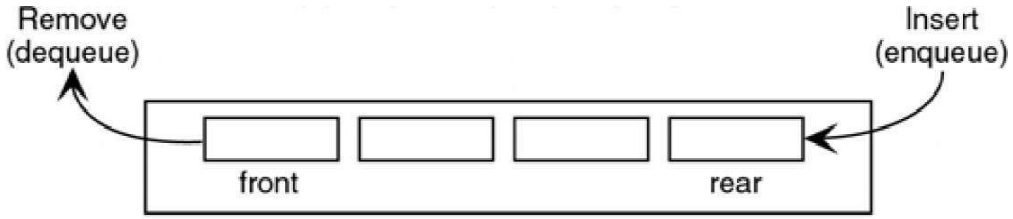
1 Understand the Queue data structure and its basic operations.

2. Understand the method of defining Queue ADT and its basic operations.

3. Learn how to create objects from an ADT and member functions are invoked.

Theory:

A queue is an ordered collection where items are removed from the front and inserted at the rear, following the First-In-First-Out (FIFO) order. The fundamental operations for a queue are "Enqueue," which adds an item to the rear, and "Dequeue," which removes an item from the front.



(b) A computer queue

Typically, a one-dimensional array is used to implement a queue, and two integer values, FRONT and REAR, track the front and rear positions in the array. When an element is removed from the queue, FRONT is incremented by one, and when an element is added to the queue, REAR is increased by one. This ensures that items are processed in the order they were added, maintaining the FIFO principle.

Algorithm:

ENQUEUE(item)

1. If (queue is full)

   Print "overflow"

2. if (First node insertion)

   Front++

3. rear++

Queue[rear]=value

DEQUEUE()

1. If (queue is empty)

   Print "underflow"

2. if(front=rear)

   Front=-1 and rear=-1

3. t = queue[front]

4. front++

5. Return t

ISEMPTY()

1. If(front = -1)then

   return 1

2. return 0

ISFULL()

1. If(rear = max)then

   return 1

2. return 0

Code:

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 10
// Changing this value will change length of array
int queue[MAX];
int front = -1, rear = -1;
void insert(void);
int delete_element(void);
int peek(void);
void display(void);
int main()
{ int option, val;
do {
printf("\n\n ***** MAIN MENU *****");
printf("\n 1. Insert an element");
printf("\n 2. Delete an element");
printf("\n 3. Peek");
printf("\n 4. Display the queue");
printf("\n 5. EXIT");
printf("\n Enter your option : ");
scanf("%d", &option);
switch(option)
{
```

```
 case 1:
 insert();
break;
case 2:
 val = delete_element();
 if (val != -1)
printf("\n The number deleted is : %d", val);
 break;
 case 3:
 val = peek(); if (val != -1)
 printf("\n The first value in queue is : %d", val);
break;
 case 4:
 display();
 break;
 }
 }
while(option != 5);
getch();
return 0;
}
void insert() {
 int num;
printf("\n Enter the number to be inserted in the queue : ");
scanf("%d", &num);
 if(rear == MAX-1)
```

```
printf("\n OVERFLOW");
else if(front == -1 && rear == -1)
front = rear = 0;
else
rear++;
 queue[rear] = num;
}
 int delete_element() {
 int val;
 if(front == -1 || front>rear) {
printf("\n  UNDERFLOW");
return -1;
 } else {
val = queue[front];
 front++;
if(front > rear)
front = rear = -1;
return val;
 }
 }
int peek() {
if(front==-1 || front>rear) {
printf("\n QUEUE IS EMPTY");
return -1;
 }
 else {
```

```
return queue[front]; }
 }
void display() {
int i;
printf("\n");
 if(front == -1 || front > rear)
printf("\n QUEUE IS EMPTY");
else {
 for(i = front;i <= rear;i++)
printf("\t %d", queue[i]);
 }
 }
```

Output:

```
***** MAIN MENU *****"
1. Insert an element
2. Delete an element
3. Peek
4. Display the queue
5. EXIT
Enter your option : 1
Enter the number to be inserted in the queue : 50
```

Conclusion:

What is the structure of queue ADT?

The Queue Abstract Data Type (ADT) is a linear data structure that follows the First-In-First-Out (FIFO) principle, meaning that the first element added to the queue is the first one to be removed. The basic operations and characteristics of a queue ADT include:

1. Enqueue:Adds an element to the back (rear) of the queue.

2. Dequeue: Removes and returns the element from the front (head) of the queue.

3. Peek (or Front): Allows you to view the element at the front of the queue without removing it.

4. IsEmpty: Checks if the queue is empty.

5. Size: Returns the number of elements currently in the queue.

A simple real-life analogy for a queue is a line of people waiting for a service, where the person who arrives first is the first to be served.

List various applications of queues?

Here are various applications of queues:

1. Print Queue: Managing print jobs in order.

2. Task Scheduling: Scheduling tasks in operating systems.

3. Breadth-First Search: Traversing graphs level by level.

4. Call Center Systems: Handling customer service calls.

5. Buffer Management: Storing data in a temporary buffer.

6. Request Handling: Managing requests in web servers.

7. Bounded Buffer: Handling data between producers and consumers.

8. Job Scheduling: Scheduling tasks in computing clusters.

9. Simulation Systems: Modeling real-world scenarios.

10. Task Queues: Managing background tasks in applications.

Where is queue used in a computer system proceesing?

Queues are used in a computer system for tasks like:

1. Task scheduling in operating systems.
2. Managing I/O operations.
3. Handling print jobs.
4. Efficient interrupt handling.
5. Buffer management.
6. Job queues in computing clusters.
7. Synchronization.
8. Message queues in distributed systems.
9. Request handling in web servers.
10. Background task management in applications.

They ensure orderly and efficient processing of tasks, data, and requests.

Experiment No.5
Implement Circular Queue ADT using array
Name: Singh Rahul Rammilan
Roll No: 56
Date of Performance:16/8/23
Date of Submission: 23/8/23
Marks:
Sign:

Experiment No. 5: Circular Queue

Aim: To Implement Circular Queue ADT using array

Objective:

Circular Queues offer a quick and clean way to store FIFO data with a maximum size

Theory:

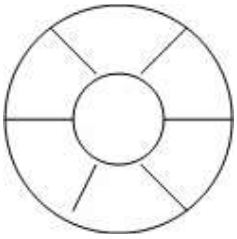
Circular queue is an data structure in which insertion and deletion occurs at an two ends rear and front respectively. Eliminating the disadvantage of linear queue that even though there is a vacant slots in array it throws full queue exception when rear reaches last element. Here in an circular queue if the array has space it never throws an full queue exception. This feature needs an extra variable count to keep track of the number of insertion and deletion in the queue to check whether the queue is full or not.Hence circular queue has better space utilization as compared to linear queue. Figure below shows the representation of linear and circular queue.

Linear queue

Front rear

0	…	…		n

Circular Queue



Algorithm

Algorithm : ENQUEUE(Item)

Input : An item is an element to be inserted in a circular queue.

Output : Circular queue with an item inserted in it if the queue has an empty slot.

Data Structure : Q be an array representation of a circular queue with front and rear pointing to the first and last element respectively.

1. If front = 0
   front = 1
   rear =1
   Q[front] = item
2. else
   next=(rear mod length)
   if next!=front then
   rear = next
   Q[rear] = item
   Else
   Print "Queue is full"
   End if
   End if
3. stop

Algorithm : DEQUEUE()

Input : A circular queue with elements.

Output :Deleted element saved in Item.

Data Structure : Q be an array representation of a circular queue with front and rear pointing to the first and last element respectively.

1. If front = 0
   Print "Queue is empty"

Exit

2. else

item = Q[front]

if front = rear then

rear = 0

front=0

else

front = front+1

end if

end if

3. stop

Code:

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
int queue[MAX];
 int front=–1, rear=–1;
 void insert(void);
int delete_element(void);
 int peek(void);
 void display(void);
 int main() {
int option, val;
clrscr();
```

```
do {
printf("\n ***** MAIN MENU *****");
printf("\n 1. Insert an element");
printf("\n 2. Delete an element");
printf("\n 3. Peek");
printf("\n 4. Display the queue");
printf("\n 5. EXIT");
printf("\n Enter your option : ");
scanf("%d", &option);
switch(option) {
case 1:
insert();
break;
case 2:
val = delete_element(); if(val!=–
1)
printf("\n The number deleted is : %d", val);
break;
case 3:
val = peek();
if(val!=–1)
printf("\n The first value in queue is : %d", val);
break;
case 4:
display();
break;
```

```
 }
 }
while(option!=5);
 getch();
return 0; }
 void insert() {
 int num;
printf("\n Enter the number to be inserted in the queue : ");
 scanf("%d", &num);
if(front==0 && rear==MAX–1)
printf("\n OVERFLOW");
else if(front==–1 && rear==–1) {
 front=rear=0;
 queue[rear]=num; }
else if(rear==MAX–1 && front!=0) {
rear=0;
 queue[rear]=num;
} else {
rear++;
queue[rear]=num;
 }
 } int delete_element() {
 int val;
 if(front==–1 && rear==– 1)
{ printf("\n UNDERFLOW");
return –1;
```

```
 }
 val = queue[front];
if(front==rear)
front=rear=–1;
else {
if(front==MAX–1)
front=0;
 else
front++;
 } return val;
 } int peek() {
 if(front==–1 && rear==– 1) {
 printf("\n QUEUE IS EMPTY");
 return –1; }
 else
{ return queue[front];
 }
 } void display() {
 int i;
 printf("\n");
 if (front ==–1 && rear= =–1)
printf ("\n QUEUE IS EMPTY");
else {
if(front<=rear;i++)
 printf("\t %d", queue[i]);
} else {
```

```
for(i=front;i<=rear;i++)
 printf("\t %d", queue[i]);
}
 }
 }
```

Output:

```
***** MAIN MENU *****
1. Insert an element
2. Delete an element
3. Peek
4. Display the queue
5. EXIT
Enter your option : 1
Enter the number to be inserted in the queue : 25
Enter your option : 2
The number deleted is : 25
Enter your option : 3
QUEUE IS EMPTY
Enter your option : 5
```

Conclusion:

Explain how Josephus Problem is resolved using circular queue and elaborate on operation used for the same.

1. Initialize a circular queue with the same number of elements as there are people in the circle.
2. Enqueue all people (or items) into the circular queue, assigning a position number to each.
3. Begin eliminating people by dequeuing every 'k-th' person from the queue.
4. Enqueue the eliminated person's position number back into the queue.
5. Repeat steps 3 and 4 until only one person (or item) remains in the queue.

The key operation used for resolving the Josephus Problem is dequeuing every 'k-th' person, effectively simulating the elimination process while maintaining the circular nature of the queue. Enqueuing the eliminated person's position number back into the queue ensures that the circle is maintained, and the process continues until the final person (or item) is left.

In summary, using a circular queue allows for an efficient and straightforward implementation of the Josephus Problem, ensuring that the 'k-th' person is eliminated in a circular fashion until only one remains.

Experiment No.6
Implement Singly Linked List ADT
Name: Singh Rahul Rammilan
Roll No:56
Date of Performance: 23/8/23
Date of Submission: 6/9/23
Marks:
Sign:

Experiment No. 6: Singly Linked List Operations

Aim: Implementation of Singly Linked List

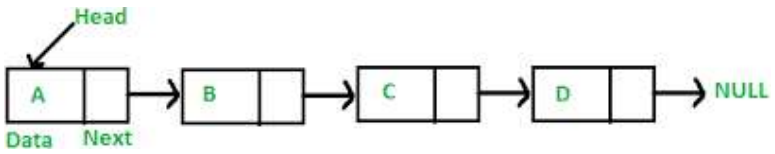
Objective:

It is used to implement stacks and queues which are like fundamental needs throughout computer science. To prevent the collision between the data in the hash map, we use a singly linked list.

Theory:

A linked list is an ordered collection of elements, known as nodes. Each node has two fields: one for data (information) and another to store the address of the next element in the list. The address field of the last node is null, indicating the end of the list. Unlike arrays, linked list elements are not stored in contiguous memory locations; instead, they are connected by explicit links, allowing for dynamic and non-contiguous memory allocation.

The structure of linked list is as shown below



Header is a node containing null in its information field and an next address field contains the address of the first data node in the list. Various operations can be performed on

singly linked lists like insertion at front, end, after a given node, before a given node deletion at front, at end and after a given node.

Algorithm

Algorithm to insert a new node at the beginning

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 7 [END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL NEXT

Step 4: SET DATA = VAL

Step 5: SET NEW_NODE -->NEXT = START

Step 6: SET START = NEW_NODE

Step 7: EXIT

Algorithm to insert a new node at the end

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 1 [END OF IF]

Step 2: SET = AVAIL

Step 3: SET AVAIL = AVAIL NEXT

Step 4: SET DATA = VAL

Step 5: SET NEW_NODE = NULL

Step 6: SET PTR = START

Step 7: Repeat Step 8 while PTR NEXT != NULL

Step 8: SET PTR = PTR NEXT [END OF LOOP]

Step 9: SET PTR--> NEXT = New_Node

Step 10: EXIT

Algorithm to insert a new node after a node that has value NUM

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 12 [END OF IF]

Step 2: SET = AVAIL

Step 3: SET AVAIL = AVAIL-->NEXT

Step 4: SET DATA = VAL

Step 5: SET PTR = START

Step 6: SET PREPTR = PTR

Step 7: Repeat Steps 8 and 9 while != NUM

Step 8: SET PREPTR = PTR

Step 9: SET PTR = PTR -->NEXT

[END OF LOOP]

Step 10 : PREPTR--> NEXT = NEW_NODE

Step 11: SET NEW_NODE NEXT = PTR

Step 12: EXIT

Algorithm to insert a new node before a node that has value NUM

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 12 [END OF IF]

Step 2: SET = AVAIL

Step 3: SET AVAIL = AVAIL-->NEXT

Step 4: SET DATA = VAL

Step 5: SET PTR = START

Step 6: SET PREPTR = PTR

Step 7: Repeat Steps 8 and 9 while PTR DATA != NUM

Step 8: SET PREPTR = PTR

Step 9: SET PTR = PTR -->NEXT

[END OF LOOP]

Step 10: PREPTR-->NEXT = NEW_NODE

Step 11: SET NEXT = PTR

Step 12: EXIT

Algorithm to delete the first node

Step 1: IF START = NULL

Write UNDERFLOW

Go to Step 5 [END OF IF]

Step 2: SET PTR = START

Step 3: SET START = START -->NEXT

Step 4: FREE PTR

Step 5: EXIT

Algorithm to delete the last node

Step 1: IF START = NULL

Write UNDERFLOW

Go to Step 8 [END OF IF]

Step 2: SET PTR = START

Step 3: Repeat Steps 4 and 5 while PTR NEXT != NULL

Step 4: SET PREPTR = PTR

Step 5: SET PTR = PTR -->NEXT [END OF LOOP]

Step 6: SET PREPTR-->NEXT = NULL

Step 7: FREE PTR

Step 8: EXIT

Algorithm to delete the node after a given node

Step 1: IF START = NULL

Write UNDERFLOW

Go to Step 1 [END OF IF]

Step 2: SET PTR = START

Step 3: SET PREPTR = PTR

Step 4: Repeat Steps 5 and 6 while PREPTR DATA != NUM

Step 5: SET PREPTR = PTR

Step 6: SET PTR = PTR--> NEXT

[END OF LOOP]

Step 7: SET TEMP = PTR

Step 8: SET PREPTR -->NEXT = PTR--> NEXT

Step 9: FREE TEMP

Step 10: EXIT

Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<malloc.h>
struct node{
int data;
struct node *next;
};
 struct node *start = NULL;
struct node *create_ll(struct node *);
struct node *display(struct node *);
struct node *insert_beg(struct node *);
struct node *insert_end(struct node *);
```

```
struct node *insert_before(struct node *);
struct node *insert_after(struct node *);
struct node *delete_beg(struct node *);
struct node *delete_end(struct node *);
struct node *delete_node(struct node *);
struct node *delete_after(struct node *);
struct node *delete_list(struct node *);
struct node *sort_list(struct node *);
int main(int argc, char *argv[]) {
int option;
 do {
 printf("\n\n *****MAIN MENU *****");
 printf("\n 1: Create a list");
 printf("\n 2: Display the list");
 printf("\n 3: Add a node at the beginning");
printf("\n 4: Add a node at the end");
printf("\n 5: Add a node before a given node");
printf("\n 6: Add a node after a given node");
printf("\n 7: Delete a node from the beginning");
printf("\n 8: Delete a node from the end");
printf("\n 9: Delete a given node");
 printf("\n 10: Delete a node after a given node");
printf("\n 11: Delete the entire list");
 printf("\n 12: Sort the list");
 printf("\n 13: EXIT");
printf("\n\n Enter your option : ");
```

```
scanf("%d", &option);
switch(option)
{
case 1:
start = create_ll(start);
printf("\n LINKED LIST CREATED");
break;
case 2:
start = display(start);
break;
case 3:
start = insert_beg(start);
break;
case 4:
start = insert_end(start);
break;
case 5:
start = insert_before(start);
break;
case 6:
start = insert_after(start);
break;
case 7:
start = delete_beg(start);
break;
case 8:
```

```
 start = delete_end(start);
 break;
case 9:
 start = delete_node(start);
 break;
case 10:
 start = delete_after(start);
break;
case 11:
 start = delete_list(start);
printf("\n LINKED LIST DELETED");
 break;
case 12:
start = sort_list(start); break;
}
}
while(option !=13);
 getch();
 return 0;
}
 struct node *create_ll(struct node *start)
{ struct node *new_node, *ptr; int num;
 printf("\n Enter -1 to end");
 printf("\n Enter the data : ");
 scanf("%d", &num);
 while(num!=-1) {
```

```
new_node = (struct node*)malloc(sizeof(struct node));
new_node -> data=num;
 if(start==NULL) { -
new_node -> next = NULL;
start = new_node;
} else {
 ptr=start;
 while(ptr->next!=NULL)
ptr=ptr->next;
 ptr->next = new_node;
 new_node->next=NULL;
 } printf("\n Enter the data : ");
scanf("%d", &num);
 } return start;
 } struct node *display(struct node *start) {
 struct node *ptr;
ptr = start;
while(ptr != NULL) {
printf("\t %d", ptr -> data);
ptr = ptr -> next;
} return start;
 } struct node *insert_beg(struct node *start) {
 struct node *new_node;
 int num;
printf("\n Enter the data : ");
 scanf("%d", &num);
```

```
new_node = (struct node*)malloc(sizeof(struct node));
new_node -> data=num;
new_node -> next = start;
start = new_node;
return start;
 } struct node *insert_end(struct node *start) {
struct node *ptr, *new_node; int num;
printf("\n Enter the data : ");
scanf("%d", &num);
 new_node = (struct node *)malloc(sizeof(struct node));
 new_node -> data = num;
new_node -> next = NULL;
ptr = start;
while(ptr -> next != NULL)
ptr = ptr -> next;
 ptr -> next = new_node;
 return start;
 } struct node *insert_before(struct node *start) {
struct node *new_node, *ptr, *preptr;
 int num, val;
 printf("\n Enter the data : ");
 scanf("%d", &num);
printf("\n Enter the value before which the data has to be inserted : ");
 scanf("%d", &val); new_node = (struct node *)malloc(sizeof(struct node));
 new_node -> data = num;
ptr = start;
```

```
while(ptr -> data != val) {
preptr = ptr;
 ptr = ptr -> next;
 } preptr -> next = new_node;
new_node -> next = ptr;
return start;
} struct node *insert_after(struct node *start) {
struct node *new_node, *ptr, *preptr;
 int num, val;
 printf("\n Enter the data : ");
 scanf("%d", &num);
 printf("\n Enter the value after which the data has to be inserted : ");
 scanf("%d", &val);
new_node = (struct node *)malloc(sizeof(struct node));
 new_node -> data = num;
ptr = start;
 preptr = ptr;
 while(preptr -> data != val) {
 preptr = ptr; ptr = ptr -> next;
}
 preptr -> next=new_node;
new_node -> next = ptr;
return start;
 } struct node *delete_beg(struct node *start)
{ struct node *ptr; ptr = start;
 start = start -> next;
```

```
 free(ptr);
return start;
 } struct node *delete_end(struct node *start) {
struct node *ptr, *preptr;
ptr = start;
 while(ptr -> next != NULL)
{ preptr = ptr;
ptr = ptr -> next;
 } preptr -> next = NULL;
free(ptr);
 return start;
 } struct node *delete_node(struct node *start) {
 struct node *ptr, *preptr; int val;
 printf("\n Enter the value of the node which has to be deleted : ");
 scanf("%d", &val);
 ptr = start;
 if(ptr -> data == val) {
 start = delete_beg(start);
 return start;
 } else {
 while(ptr -> data != val) {
preptr = ptr;
ptr = ptr -> next;
 } preptr -> next = ptr -> next;
free(ptr);
return start;
```

```
 }
 } struct node *delete_after(struct node *start) {
struct node *ptr, *preptr;
 int val;
printf("\n Enter the value after which the node has to deleted : ");
 scanf("%d", &val);
ptr = start; preptr = ptr;
while(preptr -> data != val) {
preptr = ptr; ptr = ptr -> next;
}
 preptr -> next=ptr -> next;
free(ptr);
return start;
} struct node *delete_list(struct node *start) {
 struct node *ptr;
if(start!=NULL){
 ptr=start;
while(ptr != NULL) {
printf("\n %d is to be deleted next", ptr -> data);
 start = delete_beg(ptr);
ptr = start;
 }
 }
 return start;
 }
 struct node *sort_list(struct node *start) {
```

```
struct node *ptr1, *ptr2;
int temp;
ptr1 = start;
 while(ptr1 -> next != NULL) {
 ptr2 = ptr1 -> next;
 while(ptr2 != NULL) {
 if(ptr1 -> data > ptr2 -> data) {
 temp = ptr1 -> data;
 ptr1 -> data = ptr2 -> data;
ptr2 -> data = temp; }
 ptr2 = ptr2 -> next; }
ptr1 = ptr1 -> next; }
return start;
  }
```

Output:

```
*****MAIN MENU *****
1: Create a list
2: Display the list
3: Add a node at the beginning
4: Add the node at the end
5: Add the node before a given node
6: Add the node after a given node
7: Delete a node from the beginning
8: Delete a node from the end
9: Delete a given node
10: Delete a node after a given node
11: Delete the entire list
12: Sort the list
13: Exit
Enter your option : 3
Enter your option : 73
```

Conclusion:

Write an example of stack and queue implementation using singly linked list?

```
#include <stdio.h>

#include <stdlib.h>


// Node structure for the singly linked list

struct Node {

    int data;

    struct Node* next;

};


struct Node* front = NULL;

struct Node* rear = NULL;


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;

        return;

    }
```

```
    rear->next = newNode;
    rear = newNode;
}

void dequeue() {
    if (front == NULL) {
        printf("Queue is empty.\n");
        return;
    }

    struct Node* temp = front;
    front = front->next;

    if (front == NULL) {
        rear = NULL;
    }

    free(temp);
}

int peek() {
    if (front == NULL) {
        printf("Queue is empty.\n");
```

```
        return -1;
    }
    return front->data;
}

int isEmpty() {
    return front == NULL;
}

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

    printf("Front element: %d\n", peek());
    dequeue();
    printf("Front element after dequeue: %d\n", peek());

    return 0;
}
```

Experiment No.7
Implement Circular Linked List ADT.
Name: Singh Rahul Rammilan
Roll No: 56
Date of Performance: 6/9/23
Date of Submission:13/9/23
Marks:
Sign:

Experiment No. 7: Circular Linked List Operations

Aim: Implementation of Circular Linked List ADT

Objective:

In circular linked list last node is connected to first node. On other hand circular linked list can be used to implement traversal along web pages.

Theory:

In a circular linked list, the last node contains a pointer to the first node of the list. We can have a circular singly linked list as well as a circular doubly linked list. While traversing a circular linked list, we can begin at any node and traverse the list in any one direction, forward or backward, until we reach the same node where we started. Thus, a circular linked list has no beginning and no ending.

Inserting a New Node in a Circular Linked List

Case 1: The new node is inserted at the beginning.

Case 2: The new node is inserted at the end.

Deleting a Node from a Circular Linked List

Case 1: The first node is deleted.

Case 2: The last node is deleted.

Insertion and Deletion after or before a given node is same as singly linked list.

Algorithm

Algorithm to insert a new node at the beginning

Step 1: IF AVAIL = NULL
Write OVERFLOW
Go to Step 9 [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL NEXT
Step 4: SET NEW_NODE-->DATA = VAL
Step 5: SET PTR=START
Repeat Step 6 while PTR NEXT != START
Step 6: SET PTR = PTR NEXT [END OF LOOP]

Step 7: SET NEW_NODE--> NEXT= START
Step 8: SET PTR-->NEXT = START
Step 9: SET START = NEW_NODE
Step 10: EXIT

Algorithm to insert a new node at the end

Step 1: IF AVAIL = NULL
Write OVERFLOW
Go to Step 11 [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL--> NEXT
Step 4: SET NEW_NODE -->DATA = VAL
Step 5: SET NEW_NODE-->NEXT = START
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR--> NEXT != START
Step 8: SET PTR = PTR -->NEXT [END OF LOOP]
Step 9: SET PTR -->NEXT = NEW_NODE
Step 10: EXIT

Algorithm to delete the first node

Step 1: IF START = NULL
Write UNDERFLOW
Go to Step 6 [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR--> NEXT != START
Step 4: SET PTR = PTR -->NEXT [END OF LOOP]
Step 4: SET PTR□NEXT = START -->NEXT
Step 5: FREE START
Step 6: EXIT

Algorithm to delete the last node

Step 1: IF START = NULL
Write UNDERFLOW
Go to Step 7 [END OF IF]
Step 2: SET PTR = START [END OF LOOP]
Step 3: Repeat Step 4 and Step 5 while PTR -->NEXT != START
Step 4: SET PREPTR = PTR
Step 5: SET PTR = PTR -->NEXT
Step 6: SET PREPTR-->NEXT = START
Step 7: FREE PTR
Step 8: EXIT

Code:

```
#include
#include
#include
struct node {
int data;
struct node *next;
}; struct node *start = NULL;
```

```
struct node *create_cll(struct node *);
struct node *display(struct node *);
struct node *insert_beg(struct node *);
struct node *insert_end(struct node *);
struct node *delete_beg(struct node *);
struct node *delete_end(struct node *);
struct node *delete_after(struct node *);
struct node *delete_list(struct node *);
int main() {
 int option;
 clrscr();
 do {
 printf("\n\n *****MAIN MENU *****");
printf("\n 1: Create a list");
 printf("\n 2: Display the list");
 printf("\n 3: Add a node at the beginning");
printf("\n 4: Add a node at the end");
printf("\n 5: Delete a node from the beginning");
printf("\n 6: Delete a node from the end");
printf("\n 7: Delete a node after a given node");
printf("\n 8: Delete the entire list");
printf("\n 9: EXIT");
 printf("\n\n Enter your option : ");
 scanf("%d", &option);
 switch(option) {
case 1:
```

```
 start = create_cll(start);
 printf("\n CIRCULAR LINKED LIST CREATED");
 break;
 case 2:
start = display(start);
 break;
 case 3:
 start = insert_beg(start);
 break;
 case 4:
 start = insert_end(start);
 break;
 case 5:
 start = delete_beg(start);
 break;
 case 6:
 start = delete_end(start);
 break;
case 7:
 start = delete_after(start);
break;
 case 8:
start = delete_list(start);
 printf("\n CIRCULAR LINKED LIST DELETED");
 break;
}
```

```
}while(option !=9);
 getch();
 return 0; }
 struct node *create_cll(struct node *start)
{ struct node *new_node, *ptr;
int num;
printf("\n Enter –1 to end");
 printf("\n Enter the data : ");
 scanf("%d", &num);
while(num!=–1) {
 new_node = (struct node*)malloc(sizeof(struct node));
 new_node -> data = num;
 if(start == NULL) {
new_node -> next = new_node;
 start = new_node;
 }
else
{
 ptr = start;
while(ptr -> next != start)
ptr = ptr -> next;
ptr -> next = new_node;
 new_node -> next = start;
 } printf("\n Enter the data : ");
 scanf("%d", &num);
} return start;
```

```
} struct node *display(struct node *start)
{ struct node *ptr; ptr=start;
while(ptr -> next != start) {
printf("\t %d", ptr -> data);
ptr = ptr -> next; }
 printf("\t %d", ptr -> data);
 return start; }
 struct node *insert_beg(struct node *start) {
 struct node *new_node, *ptr;
int num;
 printf("\n Enter the data : ");
 scanf("%d", &num);
new_node = (struct node *)malloc(sizeof(struct node));
 new_node -> data = num;
 ptr = start;
 while(ptr -> next != start)
 ptr = ptr -> next;
ptr -> next = new_node;
 new_node -> next = start;
 start = new_node;
return start;
} struct node *insert_end(struct node *start) {
 struct node *ptr, *new_node;
 int num;
printf("\n Enter the data : ");
scanf("%d", &num);
```

```
new_node = (struct node *)malloc(sizeof(struct node));
 new_node -> data = num;
 ptr = start;
 while(ptr -> next != start)
 ptr = ptr -> next;
 ptr -> next = new_node;
new_node -> next = start;
return start;
}
 struct node *delete_beg(struct node *start) {
 struct node *ptr;
 ptr = start;
 while(ptr -> next != start)
 ptr = ptr -> next;
ptr -> next = start -> next;
 free(start);
 start = ptr -> next;
 return start;
} struct node *delete_end(struct node *start) {
 struct node *ptr,*preptr;
ptr = start;
 while(ptr -> next != start) {
 preptr = ptr;
ptr = ptr -> next;
 }
 preptr -> next = ptr -> next;
```

```
free(ptr);
 return start;
} struct node *delete_after(struct node *start) {
 struct node *ptr, *preptr;
int val;
 printf("\n Enter the value after which the node has to deleted : ");
scanf("%d", &val);
 ptr = start;
 preptr = ptr;
 while(preptr -> data != val) {
 preptr = ptr;
ptr = ptr -> next;
} preptr -> next = ptr -> next;
 if(ptr == start) start = preptr -> next;
 free(ptr);
 return start;
 } struct node *delete_list(struct node *start) {
 struct node *ptr;
 ptr = start;
 while(ptr -> next != start)
start = delete_end(start);
free(start);
 return start;
 }
```

Output:

```
Enter the data: 4
Enter the data: -1
CIRCULAR LINKED LIST CREATED
Enter your option : 3
Enter your option : 5
Enter your option : 2
5    1    2    4
Enter your option : 9
-----------------------------------
*****MAIN MENU *****
1: Create a list
2: Display the list
3: Add a node at the beginning
-----------------------
8: Delete the entire list
9: EXIT
Enter your option : 1
Enter -1 to end
Enter the data: 1
Enter the data: 2
```

Conclusion:

Write an example of insertion and deletion in the circular linked list while traversing the web pages?

```
#include  <stdio.h>

#include <stdlib.h>

#include <string.h>


// Structure for a web page node in the circular linked list

struct WebPage {

    char title[50];

    struct WebPage* next;

};


struct WebPage* current = NULL;
```

```
// Function to insert a new web page
void insertPage(char title[]) {
    struct WebPage* newPage = (struct WebPage*)malloc(sizeof(struct WebPage));
    strcpy(newPage->title, title);

    if (current == NULL) {
        current = newPage;
        newPage->next = newPage;  // Make it point to itself in a circular list.
    } else {
        newPage->next = current->next;
        current->next = newPage;
        current = newPage;
    }
}

// Function to delete the current web page
void deletePage() {
    if (current == NULL) {
        printf("No web page to delete.\n");
        return;
    }

    struct WebPage* nextPage = current->next;
```

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

// Function to display the current web page
void displayCurrentPage() {
    if (current == NULL) {
        printf("No current web page.\n");
    } else {
        printf("Current Page: %s\n", current->title);
    }
}

int main() {
    insertPage("Home Page");
    insertPage("About Us");
    insertPage("Contact Us");
```

```
    displayCurrentPage();  // Displays "Contact Us"

    deletePage();          // Deletes "Contact Us"
    displayCurrentPage();  // Displays "About Us"

    deletePage();          // Deletes "About Us"
    displayCurrentPage();  // Displays "Home Page"

    return 0;
}
```

Experiment No.8
Implement stack ADT using Linked list
Name: Singh Rahul Rammilan
Roll No: 56
Date of Performance: 13/9/23
Date of Submission:27/9/23
Marks:
Sign:

Experiment No. 8: Stack using Linked list

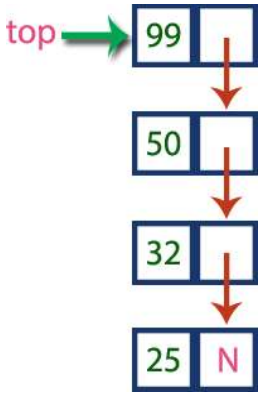
Aim: Implement Stack ADT using Linked list

Objective:

Stack can be implemented using linked list for dynamic allocation. Linked list implementation gives flexibility and better performance to the stack.

Theory:

A stack implemented using an array has a limitation in that it can only handle a fixed number of data values, and this size must be defined at the outset. This limitation makes it unsuitable for cases where the data size is unknown. On the other hand, a stack implemented using a linked list is more flexible and can accommodate an unlimited number of data values, making it suitable for variable-sized data. In a linked list-based stack, each new element becomes the 'top' element, and removal is achieved by updating 'top' to point to the previous node, effectively popping the element. The first element's "next" field should always be NULL to indicate the end of the list.



Stack Operations using Linked List

To implement a stack using a linked list, we need to set the following things before implementing actual operations.

Step 1 - Include all the header files which are used in the program. And declare all the user defined functions.

Step 2 - Define a 'Node' structure with two members data and next.

Step 3 - Define a Node pointer 'top' and set it to NULL.

Step 4 - Implement the main method by displaying Menu with list of operations and make suitable function calls in the main method.

push(value) - Inserting an element into the Stack

Step 1 - Create a newNode with given value.

Step 2 - Check whether stack is Empty (top == NULL)

Step 3 - If it is Empty, then set newNode → next = NULL.

Step 4 - If it is Not Empty, then set newNode → next = top.

Step 5 - Finally, set top = newNode.

pop() - Deleting an Element from a Stack

Step 1 - Check whether the stack is Empty (top == NULL).

Step 2 - If it is Empty, then display "Stack is Empty!!!

Step 3 - If it is Not Empty, then define a Node pointer 'temp' and set it to 'top'.

Step 4 - Then set 'top = top → next'.

Step 5 - Finally, delete 'temp'. (free(temp)).

display() - Displaying stack of elements

Step 1 - Check whether stack is Empty (top == NULL).

Step 2 - If it is Empty, then display 'Stack is Empty!!!' and terminate the function.

Step 3 - If it is Not Empty, then define a Node pointer 'temp' and initialize with top.

Step 4 - Display 'temp → data --->' and move it to the next node. Repeat the same until temp reaches to the first node in the stack. (temp → next != NULL).

Step 5 - Finally! Display 'temp → data ---> NULL'.

Code:

Code:

```
#include
#include
#include
#include
struct stack{
int data;
 struct stack *next;
 };
struct stack *top = NULL;
struct stack *push(struct stack *, int);
 struct stack *display(struct stack *);
 struct stack *pop(struct stack *);
int peek(struct stack *);
 int main(int argc, char *argv[]) {
 int val, option;
 do {
printf("\n *****MAIN MENU*****");
 printf("\n 1. PUSH");
printf("\n 2. POP");
printf("\n 3. PEEK");
printf("\n 4. DISPLAY");
 printf("\n 5. EXIT");
```

```
 printf("\n Enter your option: ");
scanf("%d", &option);
switch(option) {
 case 1:
 printf("\n Enter the number to be pushed on stack: ");
 scanf("%d", &val);
top = push(top, val);
 break;
 case 2:
 top = pop(top);
 break;
 case 3:
val = peek(top);
if (val != -1)
 printf("\n The value at the top of stack is: %d", val);
 else
 printf("\n STACK IS EMPTY");
 break;
case 4:
top = display(top);
 break; }
 }while(option != 5);
 return 0;
}
 struct stack *push(struct stack *top, int val) {
 struct stack *ptr;
ptr = (struct stack*)malloc(sizeof(struct stack));
 ptr -> data = val;
 if(top == NULL) {
ptr -> next = NULL;
top = ptr;
} else {
ptr -> next = top;
```

```
 top = ptr;
 } return top;
 } struct stack *display(struct stack *top) {
struct stack *ptr;
 ptr = top;
 if(top == NULL)
printf("\n STACK IS EMPTY");
else {
 while(ptr != NULL) {
printf("\n %d", ptr -> data);
ptr = ptr -> next;
 }
 }
 return top;
} struct stack *pop(struct stack *top) {
struct stack *ptr;
 ptr = top;
if(top == NULL)
printf("\n STACK UNDERFLOW");
 else {
top = top -> next;
 printf("\n The value being deleted is: %d", ptr -> data);
 free(ptr);
 } return top;
 } int peek(struct stack *top) {
 if(top==NULL)
return -1;
 else
return top ->data;
 }
```

Output:

```
*****MAIN MENU*****
1. PUSH
2. POP
3. Peek
4. DISPLAY
5. EXIT
Enter your option : 1
Enter the number to be pushed on stack : 100
```

Conclusion:

1. Write in detail about an application where stack is implemented as linked list?

In a programming language's runtime environment, a stack implemented as a linked list is used to manage function calls and local variables. This allows for nested function calls, proper variable scoping, and the orderly execution of functions in programs.

2. What are some real-world applications of Huffman coding, and why is it preferred in those applications?

Huffman coding is used in data compression for applications like file compression, image compression, and data transmission. It's preferred for its ability to achieve optimal compression by assigning shorter codes to more frequent data, making it efficient and saving storage and transmission costs.

3. What are the limitations and potential drawbacks of using Huffman coding in practical data compression scenarios?

Huffman coding is an efficient data compression method, but it does have limitations and potential drawbacks in practical scenarios:

1. Variable-Length Codes: While variable-length codes are efficient for compressing frequently occurring symbols, they can lead to slower decoding, as the code's length must be determined while decoding.

2. No Compression for Rare Symbols: Huffman coding doesn't compress rare symbols efficiently, potentially resulting in minimal compression gains for datasets with a wide range of symbols.

3. Memory Overhead: Storing the Huffman tree or table adds memory overhead to the compressed data, which can be significant for large datasets.

4. Compression Efficiency: Huffman coding may not always achieve the highest compression efficiency compared to more advanced compression methods like Lempel-Ziv-Welch (LZW) used in formats like ZIP or GIF.

5. Complexity: Building the Huffman tree can be computationally expensive, making it less suitable for real-time compression or scenarios with limited processing power.

6. Lossless Only: Huffman coding is suitable for lossless compression, which preserves data integrity but may not be the best choice for lossy compression, where some data loss is acceptable.

7. Lack of Adaptability:Once the Huffman tree is constructed, it remains fixed, which can be less adaptive to changes in the data distribution over time.

Despite these limitations, Huffman coding is still widely used in scenarios where lossless compression and simplicity are essential, especially when dealing with text or similar data types.

Experiment No.9
Implement Binary Search Tree ADT using Linked List.
Name: Singh Rahul Rammilan
Roll No: 56
Date of Performance: 27/9/23
Date of Submission: 4/10/23
Marks:
Sign:

Experiment No. 9: Binary Search Tree Operations

Aim : Implementation of Binary Search Tree ADT using Linked List.

Objective:

1) Understand how to implement a BST using a predefined BST ADT.

2) Understand the method of counting the number of nodes of a binary tree.

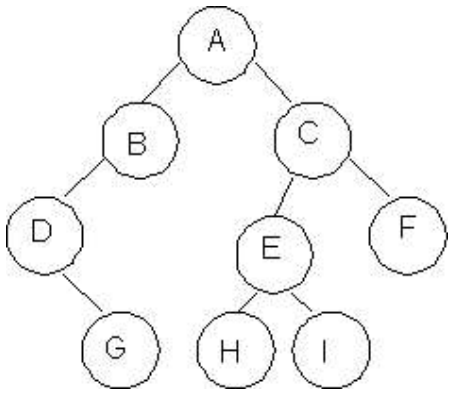
Theory:

A binary tree is a finite set of elements that is either empty or partitioned into disjoint subsets. In other words nodes in a binary tree have at most two children and each child node is referred to as left or right child.

Traversals in trees can be in one of the three ways: preorder, postorder, inorder. Preorder Traversal

Here the following strategy is followed in sequence

1. Visit the root node R
2. Traverse the left subtree of R
3. Traverse the right subtree of R



Description	**Output**
Visit Root	A
Traverse left sub tree – step to B then D	ABD
Traverse right subtree – step to G	ABDG
As left subtree is over. Visit root , which is already visited so go for right subtree	ABDGC
Traverse the left subtree	ABDGCEH
Traverse the right sub tree	ABDGCEHIF

Inorder Traversal

Here the following strategy is followed in sequence

1. Traverse the left subtree of R
2. Visit the root node R
3. Traverse the right sub tree of R

Description	**Output**
Start with root and traverse left sub tree from A-B-D	D
As D doesn't have left child visit D and go for right subtree of D which is G so visit this.	DG
Backtrack to D and then to B and visit it.	DGB

Backtrack to A and visit it	DGBA
Start with right sub tree from C-E-H and visit H	DGBAH
Now traverse through parent of H which is E and then I	DGBAHEI
Backtrack to C and visit it and then right subtree of E which is F	DGBAHEICF

Postorder Traversal

Here the following strategy is followed in sequence

1. Traverse the left subtree of R
2. Traverse the right sub tree of R
3. Visit the root node R

Description	**Output**
Start with left sub tree from A-B-D and then traverse right sub tree to get G	G
Now Backtrack to D and visit it then to B and visit it.	GD
Now as the left sub tree is over go for right sub tree	GDB
In right sub tree start with leftmost child to visit H followed by I	GDBHI
Visit its root as E and then go for right sibling of C as F	GDBHIEF
Traverse its root as C	GDBHIEFC
Finally a root of tree as A	GDBHIEFCA

Algorithm

Algorithm: PREORDER(ROOT)

Algorithm :

Function Pre-order(root)

- Start
- If root is not null then

Display the data in root

Call pre order with left pointer of root(root -> left)

Call pre order with right pointer of root(root -> right)

- Stop

Algorithm: INORDER(ROOT)

Algorithm :

Function in-order(root)

- Start
- If root is not null then

Call in order with left pointer of root (root -> left)

Display the data in root

Call in order with right pointer of root(root -> right)

- Stop

Algorithm: POSTORDER(ROOT)

Algorithm :

Function post-order (root)

- Start
- If root is not null then

Call post order with left pointer of root (root -> left)

Call post order with right pointer of root (root -> right)

Display the data in root

- Stop

Code:

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
struct node { int data; struct node *left; struct node *right; };
struct node *tree;
void create_tree(struct node *);
struct node *insertElement(struct node *, int);
void preorderTraversal(struct node *);
```

```
 void inorderTraversal(struct node *);
void postorderTraversal(struct node *);
 struct node *findSmallestElement(struct node *);
struct node *findLargestElement(struct node *);
struct node *deleteElement(struct node *, int);
struct node *mirrorImage(struct node *);
int totalNodes(struct node *);
int totalExternalNodes(struct node *);
 int totalInternalNodes(struct node *);
 int Height(struct node *);
struct node *deleteTree(struct node *);
int main() {
 int option, val;
struct node *ptr; create_tree(tree);
clrscr();
 do { printf("\n ******MAIN MENU******* \n");
 printf("\n 1. Insert Element");
 printf("\n 2. Preorder Traversal");
printf("\n 3. Inorder Traversal");
printf("\n 4. Postorder Traversal");
printf("\n 5. Find the smallest element");
printf("\n 6. Find the largest element");
printf("\n 7. Delete an element");
 printf("\n 8. Count the total number of nodes");
printf("\n 9. Count the total number of external nodes");
printf("\n 10. Count the total number of internal nodes");
```

```
 printf("\n 11. Determine the height of the tree");
printf("\n 12. Find the mirror image of the tree");
printf("\n 13. Delete the tree");
 printf("\n 14. Exit");
 printf("\n\n Enter your option : ");
scanf("%d", &option); switch(option) {
 case 1: printf("\n Enter the value of the new node : ");
scanf("%d", &val);
 tree = insertElement(tree, val);
 break;
case 2: printf("\n The elements of the tree are : \n");
preorderTraversal(tree);
 break;
case 3: printf("\n The elements of the tree are : \n");
 inorderTraversal(tree);
 break;
 case 4: printf("\n The elements of the tree are : \n");
 postorderTraversal(tree);
 break;
case 5: ptr = findSmallestElement(tree);
printf("\n Smallest element is :%d",ptr–>data);
break;
case 6: ptr = findLargestElement(tree);
 printf("\n Largest element is : %d", ptr–>data);
 break;
 case 7: printf("\n Enter the element to be deleted : ");
```

```
scanf("%d", &val);
tree = deleteElement(tree, val);
 break;
case 8: printf("\n Total no. of nodes = %d", totalNodes(tree));
 break;
case 9: printf("\n Total no. of external nodes = %d", totalExternalNodes(tree));
 break;
case 10: printf("\n Total no. of internal nodes = %d", totalInternalNodes(tree));
 break;
 case 11: printf("\n The height of the tree = %d",Height(tree));
 break;
 case 12:tree = mirrorImage(tree);
break;
 case 13: tree = deleteTree(tree);
 break;
 }
 }while(option!=14);
 getch();
 return 0;
} void create_tree(struct node *tree) {
tree = NULL;
} struct node *insertElement(struct node *tree, int val) {
struct node *ptr, *nodeptr, *parentptr;
 ptr = (struct node*)malloc(sizeof(struct node)); ptr–
>data = val;
ptr–>left = NULL;
```

```
 ptr–>right = NULL;
if(tree==NULL) { tree=ptr; tree–
>left=NULL;
 tree–>right=NULL;
} else { parentptr=NULL;
nodeptr=tree;
while(nodeptr!=NULL) {
parentptr=nodeptr;
 if(valdata) nodeptr=nodeptr–>left;
else nodeptr = nodeptr–>right;
 } if(valdata) parentptr–>left = ptr;
else parentptr–>right = ptr;
 } return tree;
 } void preorderTraversal(struct  node *tree)
if(tree != NULL)
{ printf("%d\t", tree–>data);
preorderTraversal(tree–>left);
preorderTraversal(tree–>right);
 }
 }
 void inorderTraversal(struct  node *tree) {
if(tree != NULL) {
 inorderTraversal(tree->left);
printf("%d\t", tree->data);
inorderTraversal(tree->right);
 }
```

```
}
void postorderTraversal(struct node *tree) {
 if(tree != NULL) {
 postorderTraversal(tree->left);
 postorderTraversal(tree->right);
 printf("%d\t", tree->data);
 }
 struct node *findSmallestElement(struct node *tree) {
 if( (tree == NULL) || (tree->left == NULL))
return tree;
 else
return findSmallestElement(tree ->left);
 } struct node *findLargestElement(struct node *tree)
{ if( (tree == NULL) || (tree->right == NULL))
 return tree;
 else
return findLargestElement(tree->right);
} struct node *deleteElement(struct node *tree, int val) {
 struct node *cur, *parent, *suc, *psuc, *ptr;
if(tree–>left==NULL) {
 printf("\n The tree is empty ");
 return(tree);
 } parent = tree;
cur = tree–>left;
while(cur!=NULL && val!= cur–>data) {
 parent = cur; cur = (valdata)? cur–>left:cur–>right; }
```

```
if(cur == NULL) {
printf("\n The value to be deleted is not present in the tree");
return(tree);
} if(cur–>left == NULL) ptr = cur–>right;
else
if(cur–>right == NULL)
ptr = cur–>left;
else {
// Find the in–order successor and its parent
psuc = cur;
cur = cur–>left;
while(suc–>left!=NULL) {
psuc = suc;
suc = suc–>left; }
if(cur==psuc) {
// Situation 1
suc–>left = cur–>right;
} else {
// Situation 2
suc–>left = cur–>left;
psuc–>left = suc–>right;
suc–>right = cur–>right;
} ptr = suc;
} // Attach ptr to the parent node
if(parent–>left == cur)
parent–>left=ptr;
```

```
 else parent–>right=ptr;
 free(cur);
 return tree;
 } int totalNodes(struct node *tree)
 { if(tree==NULL)
return 0;
 else
 return(totalNodes(tree–>left) + totalNodes(tree–>right) + 1);
 } int totalExternalNodes(struct node *tree) {
 if(tree==NULL)
return 0;
 else if((tree–>left==NULL) && (tree–>right==NULL))
return 1;
else
return (totalExternalNodes(tree–>left) + totalExternalNodes(tree–>right));
 } int totalInternalNodes(struct node *tree) {
 if( (tree==NULL) || ((tree–>left==NULL) && (tree–>right==NULL)))
 return 0;
 else
return (totalInternalNodes(tree–>left) + totalInternalNodes(tree–>right) + 1); }
 int Height(struct node *tree)
{ int leftheight, rightheight;
if(tree==NULL)
return 0;
else
 { leftheight = Height(tree–>left);
```

```
 rightheight = Height(tree–>right);
 if(leftheight > rightheight)
return (leftheight + 1);
 else
return (rightheight + 1);
}
 }
 struct node *mirrorImage(struct node *tree) {
struct node *ptr; if(tree!=NULL) {
mirrorImage(tree–>left);
 mirrorImage(tree–>right);
 ptr=tree–>left;
 ptr–>left = ptr–>right;
 tree–>right = ptr;
}
 } struct node *deleteTree(struct node *tree) {
 if(tree!=NULL) {
deleteTree(tree–>left);
 deleteTree(tree–>right);
 free(tree);
 }
}
```

Output:

```
*******MAIN   MENU*******
1. Insert Element
2. Preorder Traversal
3. Inorder Traversal
4. Postorder Traversal
5. Find the smallest element
6. Find the largest element
7. Delete an element
8. Count the total number of nodes
9. Count the total number of external nodes
10. Count the total number of internal nodes
11. Determine the height of the tree
12. Find the mirror image of the tree
13. Delete the tree
14. Exit
Enter your option : 1
Enter the value of the new node : 1
Enter the value of the new node : 2
Enter the value of the new node : 4
Enter your option : 3
2    1    4
Enter your option : 14
```

Conclusion:

Write a function in C program to count the number of nodes in a binary search tree?

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

// Definition for a BST node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new BST node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
```

```
}

// Function to insert a new node into the BST
struct Node* insert(struct Node* root, int data) {
    if (root == NULL)
        return createNode(data);
    if (data < root->data)
        root->left = insert(root->left, data);
    else if (data > root->data)
        root->right = insert(root->right, data);
    return root;
}

// Function to count the number of nodes in the BST
int countNodes(struct Node* root) {
    if (root == NULL)
        return 0;
    return 1 + countNodes(root->left) + countNodes(root->right);
}

int main() {
    struct Node* root = NULL;

    // Insert elements into the BST
    root = insert(root, 10);
    root = insert(root, 5);
    root = insert(root, 15);
    root = insert(root, 3);
    root = insert(root, 7);
    root = insert(root, 12);
    root = insert(root, 18);

    // Count the number of nodes in the BST
```

```
    int nodeCount = countNodes(root);
    printf("Number of nodes in the BST: %d\n", nodeCount);

    return 0;
}
```

Experiment No.10
Implementation of Graph traversal techniques - Depth First Search, Breadth First Search
Name:Singh Rahul Rammilan
Roll No:56
Date of Performance: 4/10/23
Date of Submission: 11/10/23
Marks:
Sign:

Experiment No. 10: Depth First Search and Breath First Search

Aim : Implementation of DFS and BFS traversal of graph.

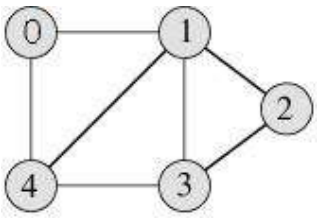
Objective:

1. Understand the Graph data structure and its basic operations.
2. Understand the method of representing a graph.
3. Understand the method of constructing the Graph ADT and defining its operations

Theory:

A graph is a collection of nodes or vertices, connected in pairs by lines referred to as edges. A graph can be directed or undirected.

One method of traversing through nodes is depth first search. Here we traverse from the starting node and proceed from top to bottom. At a moment we reach a dead end from where the further movement is not possible and we backtrack and then proceed according to left right order. A stack is used to keep track of a visited node which helps in backtracking.



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

DFS Traversal –0 1 2 3 4

Algorithm

Algorithm: DFS_LL(V)

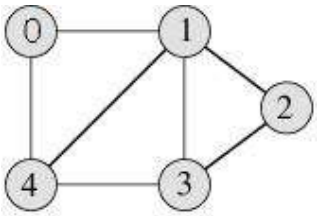
Input: V is a starting vertex

Output : A list VISIT giving order of visited vertices during traversal.

Description: linked structure of graph with gptr as pointer

1. if gptr = NULL then
   print "Graph is empty" exit
2. u=v
3. OPEN.PUSH(u)
4. while OPEN.TOP !=NULL do
   u=OPEN.POP()
   if search(VISIT,u) = FALSE then
   INSERT_END(VISIT,u)
   Ptr = gptr(u)
   While ptr.LINK != NULL do
   Vptr = ptr.LINK
   OPEN.PUSH(vptr.LABEL)
   End while
   End if
   End while
5. Return VISIT
6. Stop

BFS Traversal



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

BFS Traversal – 0 1 4 2 3

Algorithm

Algorithm: DFS()

```
i=0
count=1
visited[i]=1
print("Visited vertex  i")

repeat this till queue is empty or all nodes visited
repeat this for all nodes from first till last
if(g[i][j]!=0&&visited[j]!=1)
{
push(j)
}
i=pop()
print("Visited vertex  i")
visited[i]=1
count++
Algorithm: BFS()
i=0
count=1
visited[i]=1
print("Visited vertex  i")

repeat this till queue is empty or all nodes visited
repeat this for all nodes from first till last
if(g[i][j]!=0&&visited[j]!=1)
{
enqueue(j)
}

i=dequeue()
print("Visited vertex  i")
visited[i]=1
count++
```

Code:

BSF Code:

```
#include <stdio.h>
#define MAX 10
void breadth_first_search(int adj[][MAX],int visited[],int start) {
int queue[MAX],rear = –1,front =– 1, i;
queue[++rear] = start;
visited[start] = 1;
while(rear != front) {
start = queue[++front];
if(start == 4) printf("5\t");
else printf("%c \t",start + 65);
for(i = 0; i < MAX; i++) {
if(adj[start][i] == 1 && visited[i] == 0) {
queue[++rear] = i;
visited[i] = 1; }
}
}
}
int main() {
int visited[MAX] = {0};
int adj[MAX][MAX], i, j;
printf("\n Enter the adjacency matrix: ");
for(i = 0; i < MAX; i++)
for(j = 0; j < MAX; j++)
scanf("%d", &adj[i][j]);
```

```
 breadth_first_search(adj,visited,0);
 return 0;
}
```

DSF Code:

```
#include<stdio.h>
#define MAX 5
void depth_first_search(int adj[][MAX],int visited[],int start) {
int stack[MAX];
int top = –1, i;
 printf("%c–",start + 65);
 visited[start] = 1;
stack[++top] = start;
while(top ! = –1) {
start = stack[top];
 for(i = 0; i < MAX; i++) {
 if(adj[start][i] && visited[i] == 0) {
 stack[++top] = i;
printf("%c–", i + 65);
 visited[i] = 1; break;
}
 } if(i == MAX) top—;
}
 }
 int main() {
int adj[MAX][MAX];
 int visited[MAX] = {0}, i, j;
```

```
 printf("\n Enter the adjacency matrix: ");
for(i = 0; i < MAX; i++)
for(j = 0; j < MAX; j++)
scanf("%d", &adj[i][j]);
printf("DFS Traversal:  ");
depth_first_search(adj,visited,0);
printf("\n");
return 0;
 }
```

Output:

BFS OUTPUT:

```
Enter the adjacency matrix:
0 1 0 1 0
1 0 1 1 0
0 1 0 0 1
1 1 0 0 1
0 0 1 1 0
A B D C E
```

DFS OUTPUT:

```
Enter the adjacency matrix:
0 1 0 1 0
1 0 1 1 0
0 1 0 0 1
1 1 0 0 1
0 0 1 1 0
DFS Traversal: A -> C -> E ->
```

Conclusion:

Write the graph representation used by your program and explain why you choose that.
In this experiment, we implemented Depth First Search (DFS) and Breadth First Search (BFS) traversal algorithms for a graph. We aimed to understand the basic operations of graphs, methods of representing a graph, and how to construct a Graph ADT with its operations. -4AVARO Vidyavardhini's College of Engineering and Technology Departnment of Artificial Intelligence & Data Science For DFS, we used a stack to traverse the graph from the starting node, exploring as far as possible along each branch before backtracking. This process was implemented using a stack data structure. For BFS, we employed a queue to explore all the neighbors of a node before moving on to their neighbors. This approach ensures that nodes are visited in a level-wise fashion, which can be useful in various applications.

Write the applications of BFS and DFS other than finding connected nodes and explain how it is attained?

These algorithms have wide-ranging applications beyond finding connected nodes, including solving mazes, topological sorting, network analysis, and more. By understanding these traversal methods, we gain valuable insights into the structure and relationships within graphs, which are essential in various fields of computer science and artificial intelligence.