LOKMANYA TILAK COLLEGE OF ENGINEERING
Sector No. 4., Vikas Nagar, Koparkhairane, Navi Mumbai -400 709.

Computer Science and Engineering

(Artificial Intelligence & Machine Learning)

AY 2021-22 (Odd)

(SEM. III)

Practical file

for

Data Structure Lab

(CSL-301)

Name of Student: PRATHAMESH CHIKANKAR

Roll no. : AIMLD08

Faculty Incharge: Prof. Susmita Dutta

INDEX

SR.NO	LIST OF EXPERIMENT
1	Implement Stack using array.
2	Implement Queue using array.
3	Convert Infix expression to Postfix expression.
4	To evaluate Postfix expression using Stack ADT.
5	Implement Singly Linked List.
6	Implement Stack using linked list.
7	Implement Circular queue using Arrays.
8	Implement Queue using linked list.
9	Implement Circular linked list.
10	Implement Binary Search Tree using Linked list.

NAME : PRATHAMESH CHIKANKAR
ROLL NO. AND YEAR : AIMLD08 AND SE
BRANCH AND DIV. : CSE(AI&ML) AND D
SUBJECT : DATA STRUCTURE (DS)

EXPERIMENT NO. 1

AIM : Write a Program to Implement Stack using Arrays

THEORY : **STACK**

Stack is LIFO (Last-In, First-Out) linear data structure in which elements are inserted and removed only from one end.

Operations on Stack

A stack supports three basic operations: push, pop, and peek.

1. Push Operation

The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. Algorithm of Push operation is as follows,

ALGORITHM : PUSH

```
Step 1: IF TOP = MAX-1
            PRINT "OVERFLOW"
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

In Step 1, we first check for the OVERFLOW condition. In Step 2, TOP is incremented so that it points to the next location in the array. In Step 3, the value is stored in the stack at the location pointed by TOP.

2. Pop Operation

The pop operation is used to delete the topmost element from the stack. Algorithm of Pop operation is as follows,

ALGORITHM : POP

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
        [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

In Step 1, we first check for the UNDERFLOW condition. In Step 2, the value of the location in the stack pointed by TOP is stored in VAL. In Step 3, TOP is decremented.

3. Peek Operation

b. Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack. Algorithm of Peek operation is as follows,

ALGORITHM : PEEK

```
Step 1: IF TOP = NULL
          PRINT "STACK IS EMPTY"
Step 2: RETURN STACK[TOP]
Step 3: END
```

Peek operation first checks if the stack is empty, i.e., if TOP = NULL, then an appropriate message is printed, else the value is returned.

Assignment:

1)Write algorithm of stack implementation using array
2) PDF format of programming.
3)Answers all the following answers

1.The data structure required to check whether an expression contains balanced parenthesis is?

✓a) Stack
b) Queue
c) Array
d) Tree

. 2.The process of accessing data stored in a serial access memory is similar to manipulating data on a ------?

a) Heap
b) Binary Tree
c) Array
✓d) Stack

3.What is the result of the following operation
Top (Push (S, X))

✓a) X
b) Null
c) S
d) None

4. Which of the following statement(s) about stack data structure is/are NOT correct?

a) Stack data structure can be implemented using linked list

b) New node can only be added at the top of the stack
✓c) Stack is the FIFO data structure
d) The last node at the bottom of the stack has a NULL link

5.Consider the following operation performed on a stack of size 5.
Push(1);
Pop();
Push(2);
Push(3);
Pop();
Push(4);
Pop();
Pop();
Push(5);

After the completion of all operation, the no of element present on stack are

✓a) 1
b) 2
c) 3
d) 4

6.Which of the following operation take worst case linear time in the array implementation of stack?

a) Push
b) Pop
c) IsEmpty
✓d) None

7. The type of expression in which operator succeeds its operands is?

a) Infix Expression
b) pre fix Expression
✓c) postfix Expression
d) None

8.Which of the following application generally use a stack?

a) Parenthesis balancing program
b) Syntax analyzer in compiler
c) Keeping track of local variables at run time
✓d) All of the above

9.Consider the following array implementation of stack:

```
#define MAX 10
Struct STACK
{
Int arr [MAX];
```

```
Int top = -1;
}
```

If the array index starts with 0, the maximum value of top which does not cause stack overflow is?

✓a) 8
b) 9
c) 10
d) 11

10.Consider the usual implementation of parentheses balancing program using stack. What is the maximum number of parentheses that will appear on stack at any instance of time during the analysis of (( ) (( )) (( )))?

a) 1
b) 2
✓c) 3
d) 4

ASSIGNMENT :
Q. Write algorithm of stack implementation using array
A. :

1.Algorithm for PUSH() operation in Stack using Array:
Step 1: Start
Step 2: Declare Stack[MAX]; //Maximum size of Stack
Step 3: Check if the stack is full or not by comparing top with (MAX-1)
If the stack is full, Then print "Stack Overflow" i.e, stack is full and cannot be pushed with another element
Step 4: Else, the stack is not full
Increment top by 1 and Set, a[top] = x
which pushes the element x into the address pointed by top.
// The element x is stored in a[top]
Step 5: Stop

2.Algorithm for POP() operation in Stack using Array:
Step 1: Start
Step 2: Declare Stack[MAX]
Step 3: Push the elements into the stack
Step 4: Check if the stack is empty or not by comparing top with base of array i.e 0
If top is less than 0, then stack is empty, print "Stack Underflow"
Step 5: Else, If top is greater than zero the stack is not empty, then store the value pointed by top in a variable x=a[top] and
decrement top by 1. The popped element is x.

3.Algorithm for PEEK() operation in Stack using Arrays:
Step 1: Start
Step 2: Declare Stack[MAX]
Step 3: Push the elements into the stack
Step 4: Print the value stored in the stack pointed by top.
Step 6: Stop

PROGRAM :

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
# define MAX 5
int A[MAX];
int top=-1;
void push(int x)
{
    if(top==MAX-1)
    {
        printf("Stack overflow.\n");
    }
    else
    A[++top]=x;
}
void print()
{
    int i;
    printf("Stack: \t");
    for(i=0;i<=top;i++)
    printf("%d\t",A[i]);
    printf("\n");
}
void pop(int x)
{
    if(top==-1)
    {
        printf("Stack underflow. \n");
    }
    else
    A[top--]=x;
}
int IsEmpty()
{
    if (top==-1)
    return 1;
    else
    return 0;
}
int peek()
{
    int x=A[top];
    printf("Top most element is %d. \n",x);
}
```

```c
int main()
{
    pop(1);
    print();
    push(1);
    print();
    push(2);
    print();
    peek();
    print();
    push(3);
    print();
    push(4);
    print();
    push(5);
    print();
    push(3);
    print();
    pop(1);
    print();
    pop(1);
    print();
    peek();
    print();
    pop(1);
    print();
    pop(1);
    print();
    pop(1);
    print();
    pop(1);
    print();
    peek();
    print();
}
```

OUTPUT :

```
Stack underflow.
Stack:
Stack:  1
Stack:  1       2
Top most element is 2.
Stack:  1       2
Stack:  1       2       3
Stack:  1       2       3       4
Stack:  1       2       3
Stack:  1       2
Top most element is 2.
Stack:
Stack underflow.
Stack:
Stack underflow.
Stack:
Top most element is 0.
Stack:
```

NAME : PRATHAMESH CHIKANKAR
ROLL NO. AND YEAR : AIMLD08 AND SE
BRANCH AND DIV. : CSE(AI&ML) AND D
SUBJECT : DATA STRUCTURE (DS)

EXPERIMENT NO. 2

AIM : Write a Program to Implement Queue using Arrays

THEORY : **QUEUE**

A queue is a FIFO (First-In, First-Out) data structure in which the element that is inserted first is the first one to be taken out. The elements in a queue are added at one end called the REAR and removed from the other end called the FRONT.

Operations on Queue

1. Insertion of an Element

Insertion operation is used to add an element in the Queue. Algorithm for inserting an element is as follows,

ALGORITHM : INSERTION

```
Step 1: IF REAR = MAX-1
           Write "OVERFLOW"
           Goto Step 4
        [END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
           SET FRONT = REAR = 0
        ELSE
           SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT
```

In Step 1, we first check for the overflow condition. In Step 2, we check if the queue is empty. In case the queue is empty, then both FRONT and REAR are set tozero, so that the new value can be stored at the 0 th location. Otherwise, if the queue already has some values, then REAR is incremented so that it points to the next location in the array. In Step 3, the value is stored in the queue at the location pointed by REAR .

2. Deletion of an Element

Deletion operation is used to remove an element from the Queue. Algorithm for deleting an element is as follows,

ALGORITHM : DELETION

```
Step 1: IF FRONT = -1 OR FRONT > REAR
           Write "UNDERFLOW"
        ELSE
           SET VAL = QUEUE[FRONT]
           SET FRONT = FRONT + 1
        [END OF IF]
Step 2 : EXIT
```

In Step 1, we check for underflow condition. An underflow occurs if FRONT = -1 or FRONT > REAR. However, if queue has some values, then FRONT is incremented so that it now points to the next value

Assignment:

1)Write algorithm of stack implementation using array
2) PDF format of programming. INCLUDING OUTPUT
3)Answers all the following answers

1. Which one of the following is an application of Queue Data Structure?
(A) When a resource is shared among multiple consumers.
(B) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes
(C) Load Balancing
✓**(D)** All of the above

2.How many stacks are needed to implement a queue. Consider the situation where no other data structure like arrays, linked list is available to you.

(A) 1
✓**(B)** 2
(C) 3
(D) 4

3.How many queues are needed to implement a stack. Consider the situation where no other data structure like arrays, linked list is available to you.
1. 1
✓2. 2
3.3
4.4

4. Suppose a circular queue of capacity (n – 1) elements is implemented with an array of n elements. Assume that the insertion and deletion operation are carried out using REAR and FRONT as array index variables, respectively. Initially, REAR = FRONT = 0. The conditions to detect queue full and queue empty are
✓**(A)** Full: (REAR+1) mod n == FRONT, empty: REAR == FRONT
(B) Full: (REAR+1) mod n == FRONT, empty: (FRONT+1) mod n == REAR
(C) Full: REAR == FRONT, empty: (REAR+1) mod n == FRONT
(D) Full: (FRONT+1) mod n == REAR, empty: REAR == FRONT

5. Suppose implementation supports an instruction REVERSE, which reverses the order of elements on the stack, in addition to the PUSH and POP instructions. Which one of the following statements is TRUE with respect to this modified stack?

(A) A queue cannot be implemented using this stack.
(B) A queue can be implemented where ENQUEUE takes a single instruction and DEQUEUE takes a sequence of two instructions.
✓**(C)** A queue can be implemented where ENQUEUE takes a sequence of three instructions and DEQUEUE takes a single instruction.
(D) A queue can be implemented where both ENQUEUE and DEQUEUE take a single instruction each.

1)Write algorithm of queue implementation using array

Ans.

- Enqueue- Insert
  Step 1 − Check if the queue is full.
  Step 2 − If the queue is full, produce overflow error and exit.
  Step 3 − If the queue is not full, increment rear pointer to point the next empty space.
  Step 4 − Add data element to the queue location, where the rear is pointing.
  Step 5 − return success
- Dequeue- Delete
  Step 1 − Check if the queue is empty.
  Step 2 − If the queue is empty, produce underflow error and exit.
  Step 3 − If the queue is not empty, access the data where front is pointing.
  Step 4 − Increment front pointer to point to the next available data element.
  Step 5 − Return success

Algorithm for enqueue operation

```
procedure enqueue(data)

   if queue is full
      return overflow
   endif

   rear ← rear + 1
   queue[rear] ← data
   return true

end procedure
```

Implementation of enqueue() in C programming language −

Example

```
int enqueue(int data)
   if(isfull())
      return 0;

   rear = rear + 1;
   queue[rear] = data;

   return 1;
end procedure
```

Algorithm for dequeue operation

```
procedure dequeue

   if queue is empty
      return underflow
   end if

   data = queue[front]
   front ← front + 1
   return true

end procedure
```

Implementation of dequeue() in C programming language −

Example

```
int dequeue() {
   if(isempty())
      return 0;

   int data = queue[front];
   front = front + 1;

   return data;
}
```

- Display-
  STEP 1 - START
  STEP 2 - Check whether the queue is EMPTY.
  STEP 3 - If it is EMPTY, then display “QUEUE is EMPTY” and terminate the function.
  STEP 4 - If it is NOT EMPTY, then define an integer variable: and set ‘i=format+1’
  STEP 5 - Display ‘queue[i] value and increment ‘i’ value by one(i++). Repeat the same until(i) value reaches to rear (i<=rear)

Implementation of display() in C programming language −
Example-

```
void display()
{
    if(rear==-1)
    printf(“\n Queue is Empty!”);
    else
   {
       int i;
       printf(“Queue elements are : \n’);
       for(I=front;i<=rear;i++)
       printf(“%d\t", queue[i]);
   }
}
```

2) PDF format of programming. INCLUDING OUTPUT
Program-

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 50
void insert();
void delete();
void display();
int queue_array[MAX];
int rear=-1;
int front=-1;
int main()
{
    int choice;
    while (1)
    {
        printf("\n1.Insert element to queue \n");
        printf("2.Delete element from queue \n");
        printf("3.Display all elements of queue \n");
        printf("4.Quit \n");
        printf("Enter your choice : ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            insert();
            break;
            case 2:
            delete();
            break;
            case 3:
            display();
            break;
            case 4:
            exit(1);
            default:
            printf("\nWrong choice \n");
        }
    }
}
void insert()
{
    int item;
    if(rear==MAX-1)
    printf("\nQueue Overflow \n");
    else
    {
        if(front==-1)
        front=0;
        printf("\nInsert the element in queue : ");
        scanf("%d",&item);
        rear=rear+1;
        queue_array[rear]=item;
    }
}
void delete()
{
    if(front==-1||front>rear)
    {
        printf("\nQueue Underflow n");
        return;
    }
```

```c
    else
    {
        printf("\nElement deleted from queue is : %d \n",queue_array[front]);
        front=front+1;
    }
}
void display()
{
    int i;
    if(front==-1)
    printf("\nQueue is empty \n");
    else
    {
        printf("\nQueue is : \n");
        for(i=front;i<=rear;i++)
        printf("%d ",queue_array[i]);
    }
}
```

OUTPUT-

```
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1

Insert the element in queue : 23

1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1

Insert the element in queue : 24

1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1

Insert the element in queue : 25

1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 3

Queue is :
23 24 25
```

```
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 2

Element deleted from queue is : 23
```

```
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 3

Queue is :
24 25
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1

Insert the element in queue : 26

1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 3

Queue is :
24 25 26
1.Insert element to queue
2.Delete element from queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 4
```

NAME : PRATHAMESH CHIKANKAR
ROLL NO. AND YEAR : AIMLD08 AND SE
BRANCH AND DIV. : CSE(AI&ML) AND D
SUBJECT : DATA STRUCTURE (DS)

EXPERIMENT NO. 3

AIM : Write a Program to Convert Infix Expression to Postfix Form

THEORY : Let 'I' be an algebraic expression written in infix notation. I may contain parentheses, operands, and operators. For simplicity of the algorithm we will use only + , – , * , / , % operators. The precedence of these operators can be given as follows:

Higher priority * , / , %

Lower priority + , –

No doubt, the order of evaluation of these operators can be changed by making use of parentheses. For example, if we have an expression A + B * C , then first B * C will be done and the result will be added to A . But the same expression if written as, (A + B) * C , will evaluate A + B first and then the result will be multiplied with C.

The algorithm given below transforms an infix expression into postfix expression. The algorithm accepts an infix expression that may contain operators, operands, and parentheses. For simplicity, we assume that the infix operation contains only modulus (%), multiplication (*), division (/), addition (+), and subtraction (—) operators and that operators with same precedence are performed from left-to-right.

The algorithm uses a stack to temporarily hold operators. The postfix expression is obtained from left-to-right using the operands from the infix expression and the operators which are removed from the stack. The first step in this algorithm is to push a left parenthesis on the stack and to add a corresponding right parenthesis at the end of the infix expression. The algorithm is repeated until the stack is empty.

ALGORITHM : INFIX TO POSTFIX CONVERSION

```
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 is encountered, then
           a. Repeatedly pop from stack and add each operator (popped from the stack) to
              the postfix expression which has the same precedence or a higher precedence
              than 0
           b. Push the operator 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
```

Write a Program to Convert Infix Expression to Postfix Form
Program-

```c
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<ctype.h>
#define MAX_SIZE 100
char stack[100];
int top=-1;
void push(char x)
{
    if(top==MAX_SIZE-1)
    {
        printf("Error: stack overflow\n");
    }
    else
    top=top+1;
    stack[top]=x;
}
char pop()
{
    if (top==-1)
    printf("/nstack underflow") ;
    else
    return(stack[top--]);
}
int priority(char x)
{
    if(x=='(')
    return 0;
    if(x=='+'||x=='-')
    return 1;
    if(x=='*'||x=='/')
    return 2;
    return 0;
}
```

```c
int main()
{
    char exp[100];
    char*e,x;
    int i;
    printf("Enter the Infix expression: ");
    scanf("%s",exp);
    printf("The Postfix expression is: ");
    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-

```
Enter the Infix expression: ((A+B)-C*(D/E))+F
The Postfix expression is: AB+CDE/*-F+
PS C:\Users\Admin\Desktop\DSL\Exp.2 ig>
```

NAME : PRATHAMESH CHIKANKAR

ROLL NO. AND YEAR : AIMLD08 AND SE

BRANCH AND DIV. : CSE-(AI&ML) AND D

SUBJECT : DATA STRUCTURE (DS)

EXPERIMENT04

EXPERIMENT NO. 4

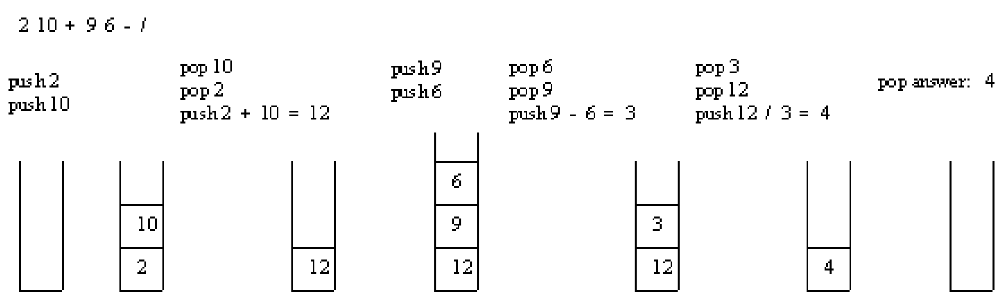
AIM : Write a Program to evaluate a postfix expression.

Theory : Algorithm for Evaluation of Postfix Expression

Create an empty stack and start scanning the postfix expression from left to right.

- If the element is an operand, push it into the stack.
- If the element is an operator O, pop twice and get A and B respectively. Calculate BOA and push it back to the stack.
- When the expression is ended, the value in the stack is the final answer.

Evaluation of a postfix expression using a stack is explained in below example:



Program :

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#define MAX 100
float st[MAX];
int top=-1;
void push(float st[],float val);
float pop(float st[]);
float evaluatePostfixExp(char exp[]);
int main()
{
    float val;
```

```
    char exp[100];
    printf("\n Enter any postfix expression : ");
    gets(exp);
    val=evaluatePostfixExp(exp);
    printf("\n Value of the postfix expression =  %.2f",val);
    return 0;
}
float evaluatePostfixExp(char exp[])
{
    int i=0;
    float op1,op2,value;
    while(exp[i]!='\0')
    {
        if(isdigit(exp[i]))
            push(st,(float)(exp[i]-'0'));
        else
        {
            op2=pop(st);
            op1=pop(st);
            switch(exp[i])
            {
            case '+':
                value=op1+op2;
                break;
            case '-':
                value=op1-op2;
                break;
            case '/':
                value=op1/op2;
                break;
            case '*':
                value=op1*op2;
                break;
            case '%':
                value=(int)op1%(int)op2;
                break;
            }
            push(st,value);
        }
        i++;
```

```
    }
    return(pop(st));
}
void push(float st[],float val)
{
    if(top==MAX-1)
        printf("\n STACK OVERFLOW!");
    else
    {
        top++;
        st[top]=val;
    }
}
float pop(float st[])
{
    float val=-1;
    if(top==-1)
        printf("\n STACK UNDERFLOW!");
    else
    {
        val=st[top];
        top--;
    }
    return val;
}
```

Output :

```
PS F:\Prathamesh\Study-Time\SE\SEM3\DSL> cd "f:\Prathamesh\Study-Time\SE\SEM3\DSL\" ;
if ($?) { gcc postfixEvaluation.c -o postfixEvaluation } ; if ($?) { .\postfixEvaluati
on }

 Enter any postfix expression : 12*3+

 Value of the postfix expression =  5.00
PS F:\Prathamesh\Study-Time\SE\SEM3\DSL>
```

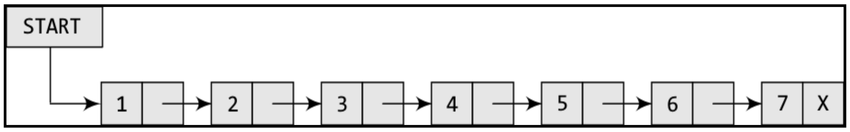
NAME : PRATHAMESH CHIKANKAR
ROLL NO. AND YEAR : AIMLD08 AND SE
BRANCH AND DIV. : CSE-(AI&ML) AND D
SUBJECT : DATA STRUCTURE (DS)

EXPERIMENT05

Experiment No. 5

<u>AIM :</u> Write a Program to Implement Singly Linked List

<u>THEORY :</u> **<u>SINGLY LINKED LIST</u>**



A singly linked list is a set of nodes where each node has two fields 'data' and 'link'. The 'data' field stores actual piece of information and 'link' field is used to point to next node. Basically 'link' field is nothing but address only.

Fig1. Singly Linked List

Operations on Singly Linked List

- **Traversing a Linked List**

Traversing a linked list means accessing the nodes of the list in order to perform some processing on them. Remember a linked list always contains a pointer variable START which stores the address of the first node of the list. End of the list is marked by storing NULL or –1 in the NEXT field of the last node. For traversing the linked list, we also make use of another pointer variable PTR which points to the node that is currently being accessed. The algorithm to traverse a linked list is shown below,

ALGORITHM : TRAVERSING

In this algorithm, we first initialize PTR with the address of START . So now, PTRpoints to the first node of the linked list. Then in Step 2, a while loop is executed which is repeated till PTR processes the last node, that is until it encounters NULL

. In Step 3, we apply the process (e.g., print) to the current node, that is, the node pointed by PTR . In Step 4, we move to the next node by making the PTR variable point to the node whose address is stored in the NEXT field.

- **Searching for a Value in a Linked List**

Searching a linked list means to find a particular element in the linked list.Algorithm to search a value in a linked list is as follows,

ALGORITHM : SEARCHNIG A NODE

In Step 1, we initialize the pointer variable PTR with START that contains the address of the first node. In Step 2, a while loop is executed which will compare every node's DATA with VAL for which the search is being made. If the search is successful, that is, VAL has been found, then the address of that node is stored in POS and the control jumps to the last statement of the algorithm. However, if the search is unsuccessful, POS is set to NULL which indicates that VAL is not present in the linked list.

- **Inserting a New Node in a Linked List**

In this section, we will see how a new node is added into an already existinglinked list. We will take four cases and then see how insertion is done in each case.

Case 1: The new node is inserted at the beginning.
Case 2: The new node is inserted at the end.
Case 3: The new node is inserted after a given node.

Case 4: The new node is
inserted before a given node.

Before we describe the algorithms to perform insertions in all these four cases, let us first discuss an important term called OVERFLOW. Overflow is a condition that occurs when AVAIL = NULL or no free memory cell is present in the system. When this condition occurs, the program must give an appropriate message.

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

ALGORITHM :INSERT A NEW NODE AT THE BEGINNING

In Step 1, we first check whether memory is available for the new node. If the free memory has exhausted, then an OVERFLOW message is printed. Otherwise, if a free memory cell is available, then we allocate space for the new node. Set its DATA part with the given VAL and the next part is initialized with the address of the first node of the list, which is stored in START. Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of the NEW_NODE. Note the following two steps:

Step 2: SET
NEW_NODE =
AVAIL Step 3:
SET AVAIL =
AVAIL ->
NEXT

These steps allocate memory for the new node. In C, there are functions like malloc(), alloc(), and calloc() which automatically do the memory allocation on behalf of the user.

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

ALGORITHM : INSERT A NEW NODE AT THE END

In Step 6, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the last node. Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node. Remember that the NEXT field of the new node contains NULL, which signifies the end of the linked list.

Case 3: The new node is inserted after a given node.

ALGORITHM : INSERT A NEW NODE AFTER A GIVEN NODE

In Step 5, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. Then we take another pointer variable PREPTR which will be used to store the address of the node preceding PTR. Initially, PREPTR is initialized to PTR. So now, PTR, PREPTR, and START are all pointing to the first node of the linked list. In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM. We need to reach this node because the new node will be inserted after this node. Once we reach this node, in Steps 10 and 11, we change the NEXT pointers in such a way that new node is inserted after the desired node.

Case 4: The new node is inserted before a given node

ALGORITHM : INSERT A NEW NODE BEFORE A GIVEN NODE

In Step 5, we take a pointer variable PTR and initialize it with START . That is, PTR now points to the first node of the linked list. Then, we take another pointer variable PREPTR and initialize it with PTR. So now, PTR, PREPTR, and START are all pointing to the first node of the linked list.

In the while loop, we traverse through the linked list to reach the node that has itsvalue equal to NUM. We need to reach this node because the new node will be inserted before this node. Once we reach this node, in Steps 10 and 11, we change the NEXT pointers in such a way that the new node is inserted before the desired node.

- **Deleting a Node from a Linked List**

In this section, we will discuss how a node is deleted from an already existing linked list. We will consider three cases and then see how deletion is done in each case.

Case 1:
The first
node is

deleted.

Case 2:

The last

node is

deleted.

Case 3: The node after a given node is deleted.

Before we describe the algorithms in all these three cases, let us first discuss an important term called UNDERFLOW. Underflow is a condition that occurs when we try to delete a node from a linked list that is empty. This happens when START

= NULL or when there are no more nodes to delete. Note that when we delete a node from a linked list, we actually have to free the memory occupied by that node. The memory is returned to the free pool so that it can be used to store other programs and data. Whatever be the case of deletion, we always change the AVAIL pointer so that it points to the address that has been recently vacated.

Case 1: The first node is deleted.

ALGORITHM : DELETE FIRST NODE

In Step 1, we check if the linked list exists or not. If START = NULL , then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm. However, if there are nodes in the linked list, then we use a pointer variable PTR that is set to point to the first node of the list. For this, we initialize PTR with START that stores the address of the first node of the list. In Step 3, START is made to point to the next node in sequence and finally the memory occupied by the node pointed by PTR (initially the first node of the list) is freed and returned to the free pool.

Case 2: The last node is deleted.

ALGORITHM :DELETE LAST NODE

In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR . Once we reach the last node and the second last node, we set the NEXT pointer of the second last node to NULL , so that it now becomes the (new) last node of the linked list. The memory of the previous last node is freed and returned back to the free pool.

Case 3: The node after a given node is deleted.

ALGORITHM : DELETE GIVEN NODE

In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR. Once we reach the node containing VAL and the node succeeding it, we set the next pointer of the node containing VAL to the address contained in next field of the node succeeding it. The memory of the node succeeding the given node is freed and returned back to the free pool.

Program :

```
#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>
struct Node
{
    int data;
    struct Node*next;
};
struct Node*head=NULL;
void printList(struct Node*head)
{

    while (head)
    {
        printf("%d ",head->data);
        head=head->next;
    }
    printf("\n");
}
struct Node*pushFront(struct Node*head,int x)
{
    struct Node*new_node=(struct Node *)malloc(sizeof(struct Node));
    new_node->data=x;
    new_node->next=head;
    printf("\nNew Node Inserted at the beginning with value %d", x);
    return new_node;
}
struct Node*popFront(struct Node*head)
{
    if(head==NULL)
    {
        printf("\nNode is empty!");
        return head;
    }
    struct Node*temp=head->next;
    free(head);
    printf("\nNode deleted from the beginning");

    return temp;
```

```
}
struct Node*pushKth(struct Node *head,int x,int n)
{
    struct Node *new_Node=(struct Node*)malloc(sizeof(struct Node));
    new_Node->data=x;
    if(n==1)
    {
        new_Node->next=head;
        head=new_Node;
        return head;
    }
    struct Node*curr=head;
    for(int i=0;(i<n-2)&&curr;i++)
    {
        curr=curr->next;
    }
    if(curr==NULL)
    {
        printf("\nThe given position beyond the limits!!\n");
        return head;
    }
    new_Node->next=curr->next;
    curr->next=new_Node;
    printf("\nNew Node inserted at given postion");
    return head;
}
struct Node*popKth(struct Node*head,int n)
{
    struct Node*curr=head;
    if (head==NULL)
    {
        return head;
    }
    if(n==1)
    {
        head=curr->next;
        free(curr);
        return head;
    }
    for(int i=0;i<n-2;i++)
```

```
    {
        curr=curr->next;
    }
    if((curr&&curr->next)==0)
    {
        printf("\nThe given position beyond the limits!!\n");
        return head;
    }
    struct Node*temp=curr->next;
    curr->next=temp->next;
    free(temp);
    printf("\nNode removed from the given postion");
    return head;
}
struct Node*pushTail(struct Node*head,int x)
{
    struct Node*temp=(struct Node*)malloc(sizeof(struct Node)); // Creting
a new Node
    temp->data=x;
    temp->next=NULL;
    if(head==NULL)         // if head is NULL we will change the head with
temp
    {
        return temp;
    }
    else
    {
        struct Node*curr=head; // curr to traverse at the end of Linked
List
        while(curr->next)  // until curr is NULL, loop will run
        {
            curr=curr->next; // Update curr next to curr
        }
        curr->next=temp; // curr is now the end of Linked List so we will
update it with temp
        return head;
    }
}
struct Node*popTail(struct Node*head)
{
```

```c
    if(head==NULL)
    {
        return NULL;
    }
    if(head->next==NULL)
    {
        free(head);
        head=NULL;
        return NULL;
    }
    struct Node*curr=head;
    while(curr->next->next)
    {
        curr=curr->next;
    }
    free(curr->next);
    curr->next=NULL;
    return head;
}
int main()
{
    int option,x,k;
    printf("\nLINKED LIST CREATED");
    do
    {
        printf("\n\n**LINKED LIST**\n");
        printf("\nSelect any one the following option : ");
        printf("\n1] Display a Linked List");
        printf("\n2] Insert a new node at the beggining");
        printf("\n3] Insert a new node at a given a postion");
        printf("\n4] Insert a new node at the end");
        printf("\n5] Delete a node at the beginning");
        printf("\n6] Delete a node at a given position");
        printf("\n7] Delete a node at the end");
        printf("\n8] EXIT\n");
        scanf("%d",&option);
        switch(option)
        {
        case 1:
            printf("\nLinked List: ");
```

```
            printList(head);
            break;
        case 2:
            printf("Enter the value: ");
            scanf("%d",&x);
            head=pushFront(head,x);
            break;
        case 3:
            printf("\nEnter the value: ");
            scanf("%d",&x);
            printf("\nEnter the position: ");
            scanf("%d",&k);
            head=pushKth(head,x,k);
            break;
        case 4:
            printf("Enter the value: ");
            scanf("%d", &x);
            head=pushTail(head,x);
            break;
        case 5:
            head=popFront(head);
            break;
        case 6:
            printf("\nEnter the position: ");
            scanf("%d",&k);
            head=popKth(head,k);
            break;
        case 7:
            head=popTail(head);
            break;
        case 8:
            break;
        default:
            printf("\nInvalid choice.");
        }
    } while(option!=8);
    return 0;
}
```

Output :

```
PS F:\Prathamesh\Study-Time\SE\SEM3\DSL> cd "f:\Prathamesh\Study-Time\SE\SEM3\DSL\" ;
if ($?) { gcc implementationOfLinkedList.c -o implementationOfLinkedList } ; if ($?) {
 .\implementationOfLinkedList }

LINKED LIST CREATED

**LINKED LIST**

Select any one the following option :
1] Display a Linked List
2] Insert a new node at the beggining
3] Insert a new node at a given a postion
4] Insert a new node at the end
5] Delete a node at the beginning
6] Delete a node at a given position
7] Delete a node at the end
8] EXIT
1

Linked List:


**LINKED LIST**

Select any one the following option :
1] Display a Linked List
2] Insert a new node at the beggining
3] Insert a new node at a given a postion
4] Insert a new node at the end
5] Delete a node at the beginning
6] Delete a node at a given position
7] Delete a node at the end
8] EXIT
2
Enter the value: 3
```

```
New Node Inserted at the beginning with value 3

**LINKED LIST**

Select any one the following option :
1] Display a Linked List
2] Insert a new node at the beggining
3] Insert a new node at a given a postion
4] Insert a new node at the end
5] Delete a node at the beginning
6] Delete a node at a given position
7] Delete a node at the end
8] EXIT
2
Enter the value: 4

New Node Inserted at the beginning with value 4

**LINKED LIST**

Select any one the following option :
1] Display a Linked List
2] Insert a new node at the beggining
3] Insert a new node at a given a postion
4] Insert a new node at the end
5] Delete a node at the beginning
6] Delete a node at a given position
7] Delete a node at the end
8] EXIT
2
Enter the value: 5

New Node Inserted at the beginning with value 5
```

```
**LINKED LIST**

Select any one the following option :
1] Display a Linked List
2] Insert a new node at the beggining
3] Insert a new node at a given a postion
4] Insert a new node at the end
5] Delete a node at the beginning
6] Delete a node at a given position
7] Delete a node at the end
8] EXIT
3

Enter the value: 5

Enter the position: 3

New Node inserted at given postion

**LINKED LIST**

Select any one the following option :
1] Display a Linked List
2] Insert a new node at the beggining
3] Insert a new node at a given a postion
4] Insert a new node at the end
5] Delete a node at the beginning
6] Delete a node at a given position
7] Delete a node at the end
8] EXIT
1

Linked List: 5 4 5 3
```

```
**LINKED LIST**

Select any one the following option :
1] Display a Linked List
2] Insert a new node at the beggining
3] Insert a new node at a given a postion
4] Insert a new node at the end
5] Delete a node at the beginning
6] Delete a node at a given position
7] Delete a node at the end
8] EXIT
5

Node deleted from the beginning

**LINKED LIST**

Select any one the following option :
1] Display a Linked List
2] Insert a new node at the beggining
3] Insert a new node at a given a postion
4] Insert a new node at the end
5] Delete a node at the beginning
6] Delete a node at a given position
7] Delete a node at the end
8] EXIT
4
Enter the value: 6
```

```
**LINKED LIST**

Select any one the following option :
1] Display a Linked List
2] Insert a new node at the beggining
3] Insert a new node at a given a postion
4] Insert a new node at the end
5] Delete a node at the beginning
6] Delete a node at a given position
7] Delete a node at the end
8] EXIT
1

Linked List: 4 5 3 6


**LINKED LIST**

Select any one the following option :
1] Display a Linked List
2] Insert a new node at the beggining
3] Insert a new node at a given a postion
4] Insert a new node at the end
5] Delete a node at the beginning
6] Delete a node at a given position
7] Delete a node at the end
8] EXIT
7
**LINKED LIST**

Select any one the following option :
1] Display a Linked List
2] Insert a new node at the beggining
3] Insert a new node at a given a postion
4] Insert a new node at the end
5] Delete a node at the beginning
6] Delete a node at a given position
7] Delete a node at the end
8] EXIT
3

Enter the value: 7

Enter the position: 4

New Node inserted at given postion

**LINKED LIST**

Select any one the following option :
1] Display a Linked List
2] Insert a new node at the beggining
3] Insert a new node at a given a postion
4] Insert a new node at the end
5] Delete a node at the beginning
6] Delete a node at a given position
7] Delete a node at the end
8] EXIT
6

Enter the position: 3
Node removed from the given postion

**LINKED LIST**

Select any one the following option :
1] Display a Linked List
2] Insert a new node at the beggining
3] Insert a new node at a given a postion
4] Insert a new node at the end
5] Delete a node at the beginning
6] Delete a node at a given position
7] Delete a node at the end
8] EXIT
8
PS F:\Prathamesh\Study-Time\SE\SEM3\DSL>
```

NAME : PRATHAMESH CHIKANKAR
ROLL NO. AND YEAR : AIMLD08 AND SE
BRANCH AND DIV. : CSE-(AI&ML) AND D
SUBJECT : DATA STRUCTURE (DS)

EXPERIMENT06

EXPERIMENT NO. 06

Aim: Write a Program in c to implement a Stack using Linked List.

Theory: Stack as we know is a Last In First Out(LIFO) data structure. It has the following operations :

push: push an element into the stack

pop: remove the last element added

top: returns the element at top of stack

Implementation of Stack using Linked List-

Stacks can be easily implemented using a linked list.

Stack is a data structure to which a data can be added using the push() method and data can be removed from it using the pop() method. With Linked list, the push operation can be replaced by the addAtFront() method of linked list and pop operation can be replaced by a function which deletes the front node of the linked list.

In this way our Linked list will virtually become a Stack with push() and pop() methods.

First we create a class node. This is our Linked list node class which will have data in it and a node pointer to store the address of the next node element.

Then we define our stack class,

Inserting Data in Stack (Linked List)

In order to insert an element into the stack, we will create a node and place it in front of the list.

Removing Element from Stack (Linked List)

In order to do this, we will simply delete the first node, and make the second node, the head of the list.

Return Top of Stack (Linked List)

In this, we simply return the data stored in the head of the list.

Program :

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
	int data;
	struct node*link;
}*top=NULL;
int isEmpty()
{
	if(top==NULL)
	{
		return 1;
	}
	else
	{
		return 0;
	}
}
void push(int item)
{
	struct node*temp;
	temp=(struct node*)malloc(sizeof(struct node));
	if(temp==NULL)
	{
		printf("Stack is Full\n");
		return;
	}
	temp->data=item;
	temp->link=top;
```

```
	top=temp;
}
int delete_node()
{
	struct node*temp;
	int item;
	if(isEmpty())
	{
		printf("Stack is Empty\n");
		exit(1);
	} temp=top;
	item=temp->data;
	top=top->link;
	free(temp);
	return item;
}
void display()
{
	struct node*ptr;
	if(isEmpty())
	{
		printf("Stack is Empty\n");
		return;
	}
	printf("Stack Elements:\n\n");
	for(ptr=top;ptr!=NULL;ptr=ptr->link)
	{
		printf(" %d\n",ptr->data);
	} printf("\n");
}
int peek()
{
	if(isEmpty())
	{
		printf("Stack is Empty\n");
		exit(1);
	} return top->data;
}
int main()
{
```

```c
	int option,element;
	while(1)
	{
		printf("1. Push an Element on the Stack\n");
		printf("2. delete_node or Delete an Element from the 
Stack\n");
		printf("3. Display Top-most item of the Stack\n");
		printf("4. Display All Element of the Stack\n");
		printf("5. Exit\n");
		printf("Enter your Option:\t");
		scanf("%d",&option);
		switch(option)
		{
			case 1:
				printf("Enter the item to be Pushed on the 
Stack:\t");
				scanf("%d",&element);
				push(element);
				break;
			case 2:
				element = delete_node();
				printf("Deleted Element:\t%d\n",element);
				break;
			case 3:
				printf("Element at the Top of 
Stack:\t%d\n",peek());
				break;
			case 4:
				display();
				break;
			case 5:
				exit(1);
			default:
				printf("Wrong Option Selected\n");
		}
	} return 0;
}
```

Output :

```
if ($?) { gcc StackADTusingLinkedList.c -o StackADTusingLinkedList } ; if ($?) { .\StackADTusingLinkedList }
1. Push an Element on the Stack
2. delete_node or Delete an Element from the Stack
3. Display Top-most item of the Stack
4. Display All Element of the Stack
5. Exit
Enter your Option:      1
Enter the item to be Pushed on the Stack:       1
1. Push an Element on the Stack
2. delete_node or Delete an Element from the Stack
3. Display Top-most item of the Stack
4. Display All Element of the Stack
5. Exit
Enter your Option:      1
Enter the item to be Pushed on the Stack:       2
1. Push an Element on the Stack
2. delete_node or Delete an Element from the Stack
3. Display Top-most item of the Stack
4. Display All Element of the Stack
5. Exit
Enter your Option:      1
Enter the item to be Pushed on the Stack:       3
1. Push an Element on the Stack
2. delete_node or Delete an Element from the Stack
3. Display Top-most item of the Stack
4. Display All Element of the Stack
5. Exit
Enter your Option:      3
Element at the Top of Stack:    3
1. Push an Element on the Stack
2. delete_node or Delete an Element from the Stack
3. Display Top-most item of the Stack
4. Display All Element of the Stack
5. Exit
```

```
Enter your Option:      2
Deleted Element:        3
1. Push an Element on the Stack
2. delete_node or Delete an Element from the Stack
3. Display Top-most item of the Stack
4. Display All Element of the Stack
5. Exit
Enter your Option:      4
Stack Elements:

 2
 1

1. Push an Element on the Stack
2. delete_node or Delete an Element from the Stack
3. Display Top-most item of the Stack
4. Display All Element of the Stack
5. Exit
Enter your Option:      5
PS F:\Prathamesh\Study-Time\SE\SEM3\DSL>
```

NAME : PRATHAMESH CHIKANKAR
ROLL NO. AND YEAR : AIMLD08 AND SE
BRANCH AND DIV. : CSE-(AI&ML) AND D
SUBJECT : DATA STRUCTURE (DS)

EXPERIMENT07

EXPERIMENT NO. 7

AIM : Write a Program to Implement Circular Queue using Arrays

THEORY : **CIRCULAR QUEUE**

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First-In, First-Out) principle and the last position is connected back to the first position.

Operations on Circular Queue

1. Insertion of an Element

Insertion operation is used to add an element in the Circular Queue. Algorithm for inserting an element is as follows,

ALGORITHM : INSERTION

```
Step 1: IF FRONT = 0 and Rear = MAX - 1
           Write "OVERFLOW"
           Goto step 4
        [End OF IF]
Step 2: IF FRONT = -1 and REAR = -1
           SET FRONT = REAR = 0
        ELSE IF REAR = MAX - 1 and FRONT != 0
           SET REAR = 0
        ELSE
           SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
```

In Step 1, we check for the overflow condition. In Step 2, we make two checks. First to see if the queue is empty, and second to see if the REAR end has already reached the maximum capacity while there are certain free locations before the FRONT end. In Step 3, the value is stored in the queue at the location pointed by REAR .

2. Deletion of an Element

Deletion operation is used to remove an element from the Circular Queue. Algorithm for deleting an element is as follows,

ALGORITHM : DELETION

```
Step 1: IF FRONT = -1
           Write "UNDERFLOW"
```

```
           Goto Step 4
        [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
           SET FRONT = REAR = -1
        ELSE
           IF FRONT = MAX -1
              SET FRONT = 0
           ELSE
              SET FRONT = FRONT + 1
           [END of IF]
        [END OF IF]
Step 4: EXIT
```

In Step 1, we check for the underflow condition. In Step 2, the value of the queue at the location pointed by FRONT is stored in VAL . In Step 3, we make two checks. First to see if the queue has become empty after deletion and second to see if FRONT has reached the maximum capacity of the queue. The value of FRONT is then updated based on the outcome of these checks.

Program :

```
#include<stdio.h>
#define MAX 6
int cqueue[MAX];
int front=-1;
int rear=-1;
void insert(int item);
void del();
void display();
int main()
{
    int choice,item;
    do
    {
        printf("1.Insert\n");
        printf("2.Delete\n");
        printf("3.Display\n");
        printf("4.Exit\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            printf("Input the element for insertion in queue : ");
            scanf("%d", &item);
            insert(item);
            break;
            case 2:
            del();
            break;
            case 3:
            display();
            break;
            case 4:
            break;
            default:
            printf("Wrong choice\n");
        }
    }
    while(choice!=4);
```

```
    return 0;
}
void insert(int item)
{
    if((front==0&&rear==MAX-1)||(front==rear+1))
    {
        printf("Queue Overflow \n");
        return;
    }
    if(front==-1)
    {
        front=0;
        rear=0;
    }
    else
    {
        if(rear==MAX-1)
        rear=0;
        else
        rear=rear+1;
    }
    cqueue[rear]=item;
}
void del()
{
    if(front==-1)
    {
        printf("Queue Underflow\n");
        return;
    }
    printf("Element deleted from queue is : %d\n",cqueue[front]);
    if(front==rear)
    {
        front=-1;
        rear=-1;
    }
    else
    {
        if(front==MAX-1)
        front=0;
```

```
        else
        front=front+1;
    }
}
void display()
{
    int front_pos=front,rear_pos=rear;
    if(front==-1)
    {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements :\n");
    if(front_pos<=rear_pos)
    while(front_pos<=rear_pos)
    {
    printf("%d ",cqueue[front_pos]);
    front_pos++;
    }
    else
    {
        while(front_pos<=MAX-1)
        {
            printf("%d ",cqueue[front_pos]);
            front_pos++;
        }
        front_pos=0;
        while(front_pos<=rear_pos)
        {
            printf("%d ",cqueue[front_pos]);
            front_pos++;
        }
    }
    printf("\n");
}
```

Output :

```
f ($?) { .\circularQueue }
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice : 1
Input the element for insertion in queue : 23
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice : 1
Input the element for insertion in queue : 45
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice : 1
Input the element for insertion in queue : 67
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice : 3
Queue elements :
23 45 67
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice : 2
Element deleted from queue is : 23
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice : 3
Queue elements :
45 67
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice : 4
PS F:\Prathamesh\Study-Time\SE\SEM3\DSL>
```

NAME : PRATHAMESH CHIKANKAR

ROLL NO. AND YEAR : AIMLD08 AND SE

BRANCH AND DIV. : CSE-(AI&ML) AND D

SUBJECT : DATA STRUCTURE (DS)

EXPERIMENT08

EXPERIMENT NO. 8

Aim: Write a Program in c for Queue using Linked List.

Theory: Like Stack, Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.

The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Operations on Queue:

Mainly the following four basic operations are performed on queue:

Enqueue: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.

Dequeue: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.

Front: Get the front item from queue.

Rear: Get the last item from queue.

In a Queue data structure, we maintain two pointers, front and rear. The front points the first item of queue and rear points to last item.

-enQueue() This operation adds a new node after rear and moves rear to the next node.

-deQueue() This operation removes the front node and moves front to the next node.

Program :

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
    int data;
    struct Node*next;
};
struct Node*head=NULL;
struct Node*tail=NULL;
struct Node*new_node(int x)
{
    struct Node*node=(struct Node*)malloc(sizeof(struct Node));
    node->data=x;
    node->next=NULL;
    return node;
}
void printQueue()
{
    struct Node*curr=head;
    printf("Queue: ");
    while(curr!=NULL)
    {
        printf("%d ",curr->data);
        curr=curr->next;
    }
    printf("\n");
}
void enqueue(int x)
{
    struct Node*newNode=new_node(x);
    if(head==NULL)
    {
        head=newNode;
        tail=newNode;
        return;
    }
    else
    {
```

```
        tail->next=newNode;
        tail=newNode;
        return;
    }
}
void deque()
{
    if(head==NULL)
    {
        printf("UnderFlow\n");
        return;
    }
    else
    {
        struct Node*temp=head;
        head=head->next;
        free(temp);
        return;
    }
}
int main()
{
    deque();
    enqueue(1);
    enqueue(2);
    printQueue();
    deque();
    enqueue(3);
    enqueue(4);
    printQueue();
    enqueue(5);
    printQueue();
    enqueue(6);
    printQueue();
    return 0;
}
```

Output :

```
PS C:\Users\Admin\Desktop\MiniProject1A> cd "f:\Prathamesh\Study-Time\SE\SEM3\DSL\" ;
if ($?) { gcc queueUsingLinkedList.c -o queueUsingLinkedList } ; if ($?) { .\queueUsin
gLinkedList }
UnderFlow
Queue: 1 2
Queue: 2 3 4
Queue: 2 3 4 5
Queue: 2 3 4 5 6
PS F:\Prathamesh\Study-Time\SE\SEM3\DSL>
```

NAME : PRATHAMESH CHIKANKAR

ROLL NO. AND YEAR : AIMLD08 AND SE

BRANCH AND DIV. : CSE-(AI&ML) AND D

SUBJECT : DATA STRUCTURE (DS)

EXPERIMENT09

EXPERIMENT NO. 9

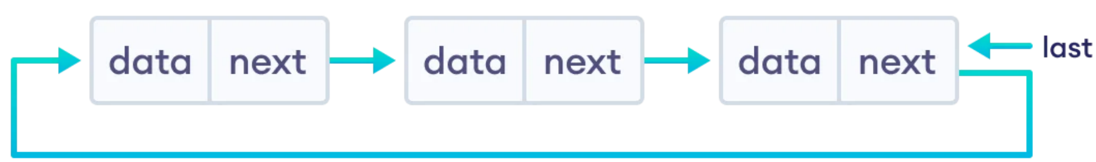
Aim: Write a Program in c for Circular Linked List.

Theory: A circular linked list is a type of linked list in which the first and the last nodes are also connected to each other to form a circle.

There are basically two types of circular linked list:

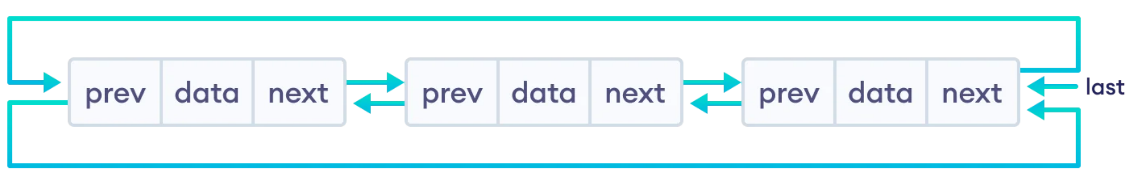
1. Circular Singly Linked List

Here, the address of the last node consists of the address of the first node.



2. Circular Doubly Linked List

Here, in addition to the last node storing the address of the first node, the first node will also store the address of the last node.



Program :

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node*next;
};
struct node*last=NULL;
void insert()
{
    int x;
    struct node*temp=(struct node *)malloc(sizeof(struct node));
    printf("\nEnter data to be inserted: \n");
    scanf("%d",&x);
    if(last==NULL)
    {
        temp->data=x;
        temp->next=temp;
        last=temp;
    }
    else
    {
        temp->data=x;
        temp->next=last->next;
        last->next=temp;
    }
}
void printList()
{
    if(last==NULL)
    printf("\nList is empty\n");
    else
    {
        struct node*temp=last->next;
        do
        {
            printf("%d-->",temp->data);
            temp=temp->next;
```

```
        }
        while(temp!=last->next);
    }
}
int main()
{
    int x,n,i;
    printf("How many nodes?\n");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        insert();
    }
    printList();
    return 0;
}
```

Output :

```
PS C:\Users\Admin\Desktop\MiniProject1A> cd "f:\Prathamesh\Study-Time\SE\SEM3\DSL\" ;
if ($?) { gcc circularLinkedList.c -o circularLinkedList } ; if ($?) { .\circularLinke
dList }
How many nodes?
5

Enter data to be inserted:
1

Enter data to be inserted:
3

Enter data to be inserted:
5

Enter data to be inserted:
2

Enter data to be inserted:
4
4-->2-->5-->3-->1-->
PS F:\Prathamesh\Study-Time\SE\SEM3\DSL>
```

NAME : PRATHAMESH CHIKANKAR
ROLL NO. AND YEAR : AIMLD08 AND SE
BRANCH AND DIV. : CSE-(AI&ML) AND D
SUBJECT : DATA STRUCTURE (DS)

EXPERIMENT10

EXPERIMENT NO. 10

Aim: Write a Program in c to implement a Binary Search Tree.

Theory: A tree is called binary when its elements have at most two children. In a binary tree, each element should have only 2 children and these are known as left and right.

Representation of Binary Tree in C:-

The value of root is NULL when a tree is empty.

It works on O(logN) for insert, search and delete operations.

A tree node includes the following parts:-

-Data

-Pointer to the left child

-Pointer to the right child

Using structures in C, you can represent a tree node.

Example:- A tree node with integer data

```
struct node
{
   int data;
   struct node *left_child;
   struct node *right_child;
};
```

In the above example, the *left_child is the pointer to the left child which can or cannot be

NULL and the *right_child is the pointer to the right child which can or cannot be NULL.

Program :

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
    struct node*lchild;
    int info;
    struct node*rchild;
};
struct node*insert(struct node*ptr,int ikey);
void display(struct node*ptr,int level,int side);
int main()
{
    struct node*root=NULL,*ptr;
    int choice,k,p,i;
    while(1)
    {
        printf("\n");
        printf("1.Insert\n");
        printf("2.Display\n");
        printf("3.Quit\n");
        printf("\nEnter your choice : ");
        scanf("%d", &choice);
        switch (choice)
        {
        case 1:
            printf("\nEnter the key to be inserted : ");
            scanf("%d",&k);
            root=insert(root, k);
            break;
        case 2:
            printf("\n");
            display(root,0,1);
            printf("\n");
            break;
        case 3:
            exit(1);
        default:
            printf("\nWrong choice\n");
```

```c
        }
    }
    return 0;
}
struct node*insert(struct node*ptr,int ikey)
{
    if (ptr==NULL)
    {
        ptr=(struct node *)malloc(sizeof(struct node));
        ptr->info=ikey;
        ptr->lchild=NULL;
        ptr->rchild=NULL;
    }
    else if(ikey<ptr->info)
        ptr->lchild=insert(ptr->lchild,ikey);
    else if(ikey>ptr->info)
        ptr->rchild=insert(ptr->rchild,ikey);
    else
        printf("\nDuplicate key\n");
    return ptr;
}
void display(struct node*ptr,int level,int side)
{
    int i,p=0;
    if(ptr==NULL)
        return;
    level++;
    display(ptr->rchild,level,1);
    printf("\n");
    for(i=0;i<level;i++)
    {
        printf("\t");
    }
    if(side==1)
        printf("/ ");
    else
        printf("\\ ");
    printf("%d",ptr->info);
    display(ptr->lchild,level,0);
}
```

Output :

```
if ($?) { gcc implementationOfBST.c -o implementationOfBST } ; if ($?) { .\implementat
ionOfBST }

1.Insert
2.Display
3.Quit

Enter your choice : 1

Enter the key to be inserted : 1

1.Insert
2.Display
3.Quit

Enter your choice : 1

Enter the key to be inserted : 4

1.Insert
2.Display
3.Quit

Enter your choice : 1

Enter the key to be inserted : 7

1.Insert
2.Display
3.Quit

Enter your choice : 1

Enter the key to be inserted : 2
```

```
1.Insert
2.Display
3.Quit

Enter your choice : 1

Enter the key to be inserted : 5

1.Insert
2.Display
3.Quit
```

```
Enter your choice : 1

Enter the key to be inserted : 8

1.Insert
2.Display
3.Quit

Enter your choice : 1

Enter the key to be inserted : 3

1.Insert
2.Display
3.Quit

Enter your choice : 1

Enter the key to be inserted : 6

1.Insert
2.Display
3.Quit

Enter your choice : 1

Enter the key to be inserted : 9

1.Insert
2.Display
3.Quit

Enter your choice : 2
```

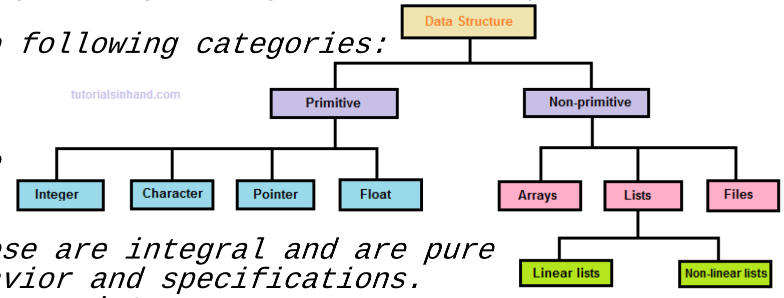
```
                                        / 9
                                / 8
                        / 7
                                        / 6
                                \ 5
                / 4
                                / 3
                        \ 2
        / 1

1.Insert
2.Display
3.Quit

Enter your choice : 3
PS F:\Prathamesh\Study-Time\SE\SEM3\DSL>
```

NAME : PRATHAMESH CHIKANKAR
ROLL NO. AND YEAR : AIMLD08 AND SE
BRANCH AND DIV. : CSE(AI&ML) AND D
SUBJECT : DATA STRUCTURE (DS)

Assignment No. 1
Last Date or submission: 10/10/21

Q.1. Explain different types of data structures with examples.

Ans. *A data structure is a particular way of organizing data in a computer so that it can be used effectively.*
Data structure can be divided into following categories:
1.Primitive data structure
2.Non-primitive data structure



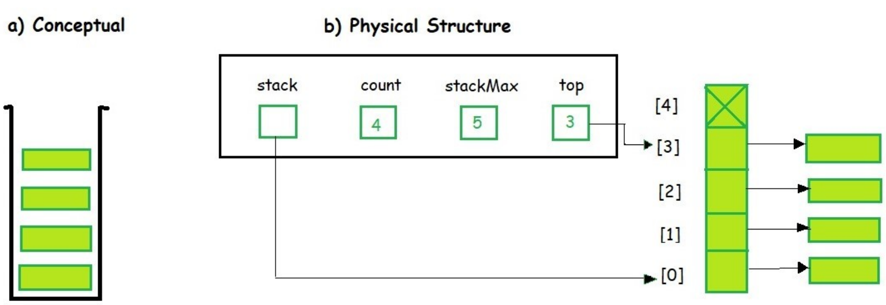
Primitive data structure
These are the structures which are supported at the machine level, they can be used to make non-primitive data structures. These are integral and are pure in form. They have predefined behavior and specifications.
Examples: Integer, float, character, pointers.
The pointers, however don't hold a data value, instead, they hold memory addresses of the data values. These are also called the reference data types.

Non-primitive data structure
The non-primitive data structures cannot be performed without the primitive data structures. Although, they too are provided by the system itself yet they are derived data structures and cannot be formed without using the primitive data structures.
The important non-primitive data structure are:
Examples: Arrays, Lists, Files.
Lists may further be classified as:
Linear lists → Stack, Queue
Non-linear lists → Graph, Tree.

Q.2. What is Abstract Data Type? Write ADT of stack and state applications of stack.

Ans. *Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of value and a set of operations.*
The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.

ADT of stack
In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored. The program allocates memory for the data and address is passed to the stack ADT.The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack. The stack head structure also contains a pointer to top and count of number of entries currently in stack.




```
//Stack ADT Type Definitions
typedef struct node
{
   void *DataPtr;
   struct node *link;
}StackNode;
typedef struct
{
   int count;
   StackNode *top;
} STACK;
```

A Stack contains elements of the same type arranged in sequential order. All operations take place at a single end that is top of the stack and following operations can be performed:
push() – Insert an element at one end of the stack called top.
pop() – Remove and return the element at the top of the stack, if it is not empty.
peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
size() – Return the number of elements in the stack.
isEmpty() – Return true if the stack is empty, otherwise return false.
isFull() – Return true if the stack is full, otherwise return false.

Applications of stack

In a stack, only limited operations are performed because it is restricted data structure. The elements are deleted from the stack in the reverse order.
Following are some of the important applications of a Stack data structure:

1. Stacks can be used for expression evaluation.
2. Stacks can be used to check parenthesis matching in an expression.
3. Stacks can be used for Conversion from one form of expression to another.
4. Stacks can be used for Memory Management.
5. Stack data structures are used in backtracking problems.

Q.3. Explain infix, postfix and prefix expressions with example.

Ans. An Expression is a combination of symbols that can be numbers (constants), variables, operations, symbols of grouping and other punctuation written is a specific format/way.

Depending on how the expression is written, we can classify it into 3 categories –

1) Prefix –
An expression is called the prefix expression if the operator appears in the expression before the operands. Simply of the form (operator operand1 operand2).
E.g. +AB

2) Infix –
An expression is called the infix expression if the operator appears in the expression in between the operands. Simply of the form (operand1 operator operand2).
E.g. A+B

3) Postfix –
An expression is called the postfix expression if the operator appears in the expression after the operands. Simply of the form (operand1 operand2 operator).
E.g. AB+

Order of Operations(Operator Precedence) –
1) Parentheses – {}, [], ()
2) Exponents(Right to Left) – A^B, 2^3^4
3) Multiplication & Division(Left to Right) – A*B/C
4) Addition & Subtraction(Left to Right) – A + B – C

Associativity –
Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a + b − c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators.

Q.4. Evaluate the following expression using stack
ABC*DEF^/G*-H*+
where A = 6, B=1, C= 4, D = 16, E=2, F= 3, G = 2, H = 5

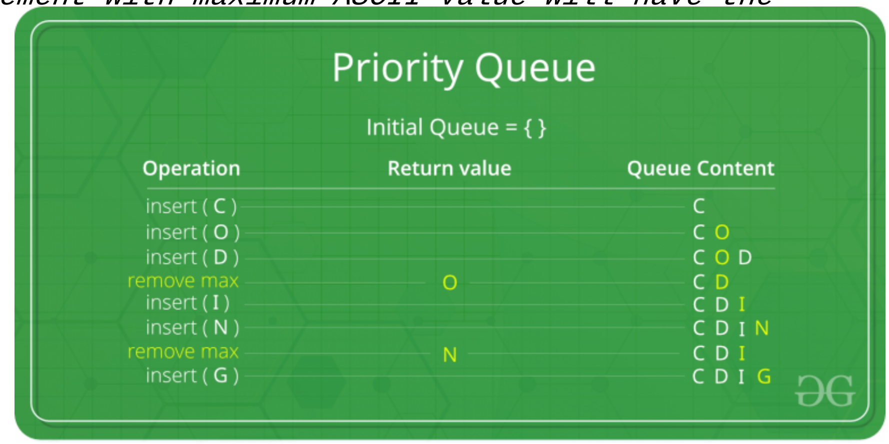
Ans.

Character	Stack
A	A
B	AB
C	ABC
*	A(B*C)
D	A(B*C)D
E	A(B*C)DE
F	A(B*C)DEF
^	A(B*C)DE^F
/	A(B*C)D/E^F
G	A(B*C)D/E^FG
*	A(B*C)D/E^F*G
-	A(B*C)-D/E^F*G
H	A(B*C)-D/E^F*GH
*	A(B*C)-D/E^F*G*H
+	A+B*C-D/E^F*G*H

Infix expression is A+(B*C-(D/E^F)*G)*H
Substituting all the values we get,
= 6+(1*4-(16/2^3)*2)*5
= 6+(1*4-(16/8)*2)*5
= 6+(1*4-(2*2))*5
= 6+0*5
= 6+0
= 6

Q.5. Write a note on Priority Queue.

Ans. Priority Queue is an extension of queue with following properties.
Every item has a priority associated with it.
An element with high priority is dequeued before an element with low priority.
If two elements have the same priority, they are served according to their order in the queue.
In the below priority queue, element with maximum ASCII value will have the highest priority.



Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. So we're assigned priority to item based on its key value. Lower the value, higher the priority.

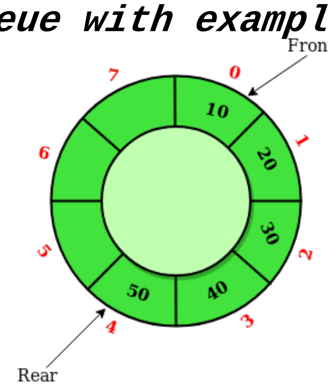
A typical priority queue supports following operations.
insert(item, priority): Inserts an item with given priority.
getHighestPriority(): Returns the highest priority item.
deleteHighestPriority(): Removes the highest priority item.
How to implement priority queue?
Using Array: A simple implementation is to use array of following structure.

```
struct item
{
   int item;
   int priority;
}
```

insert() operation can be implemented by adding an item at end of array in O(1) time.
getHighestPriority() operation can be implemented by linearly searching the highest priority item in array. This operation takes O(n) time.
deleteHighestPriority() operation can be implemented by first linearly searching an item, then removing the item by moving all subsequent items one position back.
We can also use Linked List, time complexity of all operations with linked list remains same as array. The advantage with linked list is deleteHighestPriority() can be more efficient as we don't have to move items.

Q.6. Explain Circular queue and Double ended queue with example.

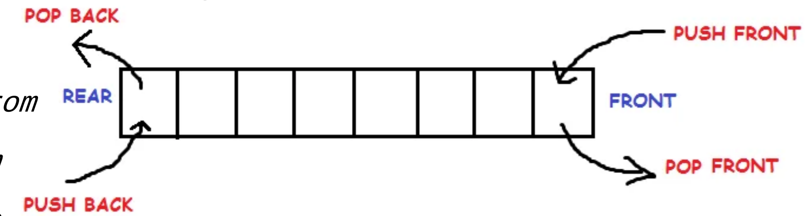
Ans. Circular Queue *is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called 'Ring Buffer'.*



In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.

Operations on Circular Queue:
Front: Get the front item from queue.
Rear: Get the last item from queue.
enQueue(value) This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.
Check whether queue is Full –
Check ((rear == SIZE-1 && front == 0) || (rear == front-1)).
If it is full then display Queue is full. If queue is not full then, check if(rear == SIZE – 1 && front != 0) if it is true then set rear=0 and insert element. deQueue() This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.
Check whether queue is Empty means check (front==-1).
If it is empty then display Queue is empty. If queue is not empty then step 3
Check if (front==rear) if it is true then set front=rear= -1 else check if (front==size-1), if it is true then set front=0 and return the element.

Deque or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends.
Operations on Deque:
Mainly the following four basic operations are performed on queue:
insertFront(): Adds an item at the front of Deque.
insertLast(): Adds an item at the rear of Deque.
deleteFront(): Deletes an item from front of Deque.
deleteLast(): Deletes an item from rear of Deque.
In addition to above operations, following operations are also supported.
getFront(): Gets the front item from queue.
getRear(): Gets the last item from queue.
isEmpty(): Checks whether Deque is empty or not.
isFull(): Checks whether Deque is full or not.



Q.7. Write a C program to implement singly linked list. Provide the following operations:
i) Insert at beginning
ii) Insert at location
iii) Delete from beginning
iv) Delete from location

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
struct Node
{
    int data;
    struct Node *next;
};
struct Node *head = NULL;
void printList(struct Node *head)
{
    while (head)
    {
        printf("%d ", head->data);
        head = head->next;
    }
    printf("\n");
}
struct Node *pushFront(struct Node *head, int x)
{
    struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
    new_node->data = x;
    new_node->next = head;
    printf("\nNew Node Inserted at the beginning with value %d", x);
    return new_node;
}
struct Node *popFront(struct Node *head)
{
    if (head == NULL)
    {
        printf("\nNode is empty");
        return head;
    }
    struct Node *temp = head->next;
    free(head);
    printf("\nNode deleted from the beginning");

    return temp;
}
```

```
struct Node *pushKth(struct Node *head, int x, int n)
{
    struct Node *new_Node = (struct Node *)malloc(sizeof(struct Node));
    new_Node->data = x;
    if (n == 1)
    {
        new_Node->next = head;
        head = new_Node;
        return head;
    }
    struct Node *curr = head;
    for (int i = 0; (i < n - 2) && curr; i++)
    {
        curr = curr->next;
    }
    if (curr == NULL)
    {
        printf("\nThe given position beyond the limits\n");
        return head;
    }
    new_Node->next = curr->next;
    curr->next = new_Node;
    printf("\nNew Node inserted at given postion");
    return head;
}
struct Node *popKth(struct Node *head, int n)
{
    struct Node *curr = head;
    if (head == NULL)
    {
        return head;
    }
    if (n == 1)
    {
        head = curr->next;
        free(curr);
        return head;
    }
    for (int i = 0; i < n - 2; i++)
    {
        curr = curr->next;
    }
    if ((curr && curr->next) == 0)
    {
        printf("\nThe given position beyond the limits\n");
        return head;
    }
    struct Node *temp = curr->next;
    curr->next = temp->next;
    free(temp);
    printf("\nNode removed from the given postion");
    return head;
}
int main()
{
    int option, x, k;
    printf("\nLINKED LIST CREATED");
    do
    {
        printf("\n\n**LINKED LIST**\n");
        printf("\nSelect any one the following option : ");
        printf("\n1] Display a Linked List");
        printf("\n2] Insert a new node at the beggining");
        printf("\n3] Insert a new node at a given a postion");
        printf("\n4] Delete a node at the beginning");
        printf("\n5] Delete a given node");
        printf("\n6] EXIT\n");
        scanf("%d", &option);
```

```c
        switch (option)
        {
        case 1:
            printf("\nLinked List: ");
            printList(head);
            break;
        case 2:
            printf("Enter the value: ");
            scanf("%d", &x);
            head = pushFront(head, x);
            break;
        case 3:
            printf("\nEnter the value: ");
            scanf("%d", &x);
            printf("\nEnter the position: ");
            scanf("%d", &k);
            head = pushKth(head, x, k);
            break;
        case 4:
            head = popFront(head);
            break;
        case 5:
            printf("\nEnter the position: ");
            scanf("%d", &k);
            head = popKth(head, k);
            break;
        case 6:
            break;
        default:
            printf("\nInvalid choice.");
        }
    } while (option != 6);
    return 0;
}
```

Q.8. Write a C program to implement Queue using Single linked list.

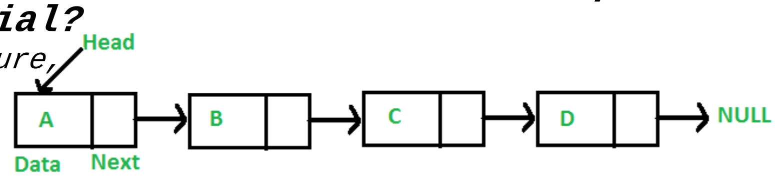
Program.

```c
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *next;
};
struct node *front;
struct node *rear;
void insert();
void delete();
void display();
void main ()
{
    int choice;
    while(choice != 4)
    {
        printf("\n**************************Main Menu******************************\n");
        printf("\n====================================================================\n");
        printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
        printf("\nEnter your choice ?");
        scanf("%d",& choice);
        switch(choice)
        {
            case 1:
            insert();
            break;
            case 2:
            delete();
            break;
            case 3:
            display();
            break;
            case 4:
            exit(0);
            break;
            default:
            printf("\nEnter valid choice??\n");
        }
```

```c
    }
}
void insert()
{
    struct node *ptr;
    int item;

    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW\n");
        return;
    }
    else
    {
        printf("\nEnter value?\n");
        scanf("%d",&item);
        ptr -> data = item;
        if(front == NULL)
        {
            front = ptr;
            rear = ptr;
            front -> next = NULL;
            rear -> next = NULL;
        }
        else
        {
            rear -> next = ptr;
            rear = ptr;
            rear->next = NULL;
        }
    }
}
void delete ()
{
    struct node *ptr;
    if(front == NULL)
    {
        printf("\nUNDERFLOW\n");
        return;
    }
    else
    {
        ptr = front;
        front = front -> next;
        free(ptr);
    }
}
void display()
{
    struct node *ptr;
    ptr = front;
    if(front == NULL)
    {
        printf("\nEmpty queue\n");
    }
    else
    {   printf("\nprinting values .....\n");
        while(ptr != NULL)
        {
            printf("\n%d\n",ptr -> data);
            ptr = ptr -> next;
        }
    }
}
```

Q.9. Give applications of linked list. How it can be used to perform additive operation on polynomial?

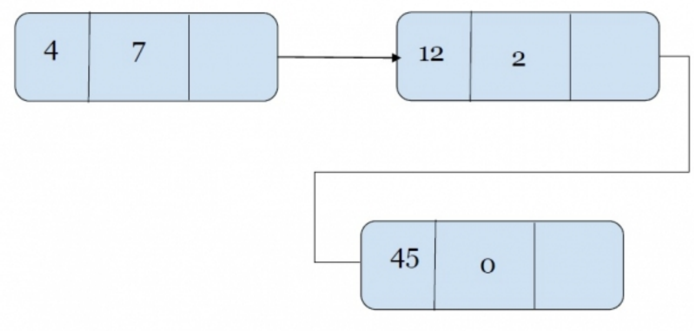
Ans. A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the image:



Applications of linked list in computer science –

1. Implementation of stacks and queues
2. Implementation of graphs : Adjacency list representation of graphs is most popular which is uses linked list to store adjacent vertices.
3. Dynamic memory allocation : We use linked list of free blocks.
4. Maintaining directory of names
5. Performing arithmetic operations on long integers
6. Manipulation of polynomials by storing constants in the node of linked list
7. representing sparse matrices

Polynomial is a mathematical expression that consists of variables and coefficients.
for example x^2 - 4x + 7
In the Polynomial linked list, the coefficients and exponents of the polynomial are defined as the data node of the list.



This is how a linked list represented polynomial looks like.

For adding two polynomials that are stored as a linked list. We need to add the coefficients of variables with the same power. In a linked list node contains 3 members, coefficient value link to the next node.
a linked list that is used to store Polynomial looks like −
Polynomial : 4x7 + 12x2 + 45
Adding two polynomials that are represented by a linked list.
We check values at the exponent value of the node. For the same values of exponent, we will add the coefficients.
Example,
Input :p1= 13x8 + 7x5 + 32x2 + 54
p2= 3x12 + 17x5 + 3x3 + 98
Output : 3x12 + 13x8 + 24x5 + 3x3 + 32x2 + 152
Explanation − For all power, we will check for the coefficients of the exponents that have the same value of exponents and add them.
The return the final polynomial.
Algorithm
Input − polynomial p1 and p2 represented as a linked list.
Step 1: loop around all values of linked list and follow step 2& 3.
Step 2: if the value of a node’s exponent. is greater copy this node to result node and head towards the next node.
Step 3: if the values of both node’s exponent is same add the coefficients and then copy the added value with node to the result.
Step 4: Print the resultant node.