DEPARTMENT OF COMPUTER ENGINEERING

LABORATORY MANUAL

DATA STRUCTURES LABORATORY

(BTCOL306)

Shri Vile Parle Kelavani Mandal's
INSTITUTE OF TECHNOLOGY DHULE (M.S.)

SHRI VILE PARLE KELAVANI MANDAL'S

INSTITUTE OF TECHNOLOGY, DHULE

Survey No. 499, Plot No. 02, Behind Gurudwara, Mumbai-Agra National Highway, Dhule-424001, Maharashtra, India. Office Phone: 02562-297801 / 297601, Fax : 02562-297801,
Mail: IOTDhule@svkm.ac.in

Vision of Institute

To be a socially sensitive engineering institute of excellence adding value to the nation.

Mission of Institute

1. To provide resources of excellence with a focus on nurturing and developing the society.
2. To strive to be an institute of global recognition.

Vision of Department

We envision a globally recognized and innovative computer engineer who meets socio-economical and industrial needs.

Mission of Department

1. To empower students with comprehensive knowledge of Computer Engineering to be successful professionals.
2. To ensure quality education that prepares students for careers in industry and higher education.
3. To develop leadership skills in students while instilling strong

Program Educational Objectives (PEOs) of Department

PEO I: Graduates will have a successful professional career while maintaining strong ethical values.

PEO II: Graduates will demonstrate knowledge and technological skills to resolve real life problems.

PEO III: Graduates will progress as professionals, researchers, or entrepreneurs who continues to learn and adopt emerging technology.

COMPUTER ENGINEERING DEPARTMENT

Program Outcomes

Engineering Graduates will be able to:

1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

3. Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4. Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. Ethics: Apply ethical principles and commit to professional ethics, responsibilities, and norms of the engineering practice.

9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

COMPUTER ENGINEERING DEPARTMENT

Program Specific Outcomes (PSO) addressed by the Course:

A graduate of the Computer Engineering Program will demonstrate-

PSO1: Professional Skills-To gain the ability to comprehend, analyze, design and implement computer programs in the fields of computer algorithms, web development, data science, computer network and security, software design, system software, cloud computing and allied fields.

PSO2: Problem-Solving Skills- Capability to provide computer based solutions to a variety of problems by applying standard practices, problem solving strategies and methodologies.

PSO3: Professional Career - The ability to create an innovative career path by utilizing modern computer tools and technologies.

COMPUTER ENGINEERING DEPARTMENT

Program Specific Outcomes (PSO) addressed by the Course:

A graduate of the Computer Engineering Program will demonstrate-

PSO1: Professional Skills-The ability to understand, analyze and develop computer programs in the areas related to algorithms, system software, multimedia, web design, data science, and networking for efficient design of computer-based systems.

PSO2: Problem-Solving Skills- The ability to apply standard practices and strategies in software project development using open-ended programming environments to deliver advanced computing systems.

PSO3: Professional Career and Entrepreneurship- The ability to employ modern computer languages, environments, and platforms in creating innovative career paths to be an entrepreneur, and a zest for higher studies and research.

Dr. Babasaheb Ambedkar Technological University

Syllabus & Scheme

Course Code	Course Title	Weekly Teaching hrs.	Evaluation Scheme			Credit
				CA	ESE	
BTCOL306	Data Structure Laboratory	**2**		**60**	**40**	**2**

BTCOL306 Data Structure Laboratory

The objective of this lab is to teach students various data structures and to explain them algorithms for performing various operations on these data structures. This lab complements the data structures course. Students will gain practical knowledge by writing and executing programs in C using various data structures such as arrays, linked lists, stacks, queues, trees, graphs and search trees.

Shri Vile Parle Kelavani Mandal's
Institute of Technology, Dhule.
www.svkm-iot.ac.in
Survey No. 499, Plot No.2, Behind Gurudwara, Mumbai-Agra Road, Dhule(M.S.)

DEPARMENT OF COMPUTER ENGG.

LABORATORY : BTCOL306 Data Structure Laboratory | **SEMESTER : 3rd Sem**

Master List of Practical

SR. NO.	PRACTICAL DESCRIPTION	Page No.
1	Write a program to implement stack using arrays.	
2	Write a program to evaluate a given postfix expression using stacks	
3	Write a program to convert a given infix expression to postfix form using stacks.	
4	Write a program to implement circular queue using arrays.	
5	Write a program to implement double ended queue (dequeue) using arrays.	
6	Write a program to implement a stack using two queues such that the push operation runs in constant time and the pop operation runs in linear time.	
7	Write a program to implement a stack using two queues such that the push operation runs in linear time and the pop operation runs in constant time.	
8	Write a program to implement a queue using two stacks such that dequeue operation runs in constant time and dequeue operation runs in linear time.	
9	Write programs to implement the following data structures: (a) Single linked list (b)Double linked list.	
10	Write a program to implement a stack using a linked list such that the push and pop operations of stack still take O(1)time.	
11	Write a program to create a binary search tree (BST) by considering the keys in given order and perform the following operations on it. (a) Minimum key (b) Maximum key (c) Search for a given key (d) Find predecessor of a node (e) Find successor of a node (f) delete a node with given key.	
12	Write a program to construct an AVL tree for the given set of keys. Also write function for deleting a key from the given AVL tree.	
13	Write a program to implement hashing with (a) Separate Chaining and (b) Open addressing methods.	

14	Implement the following sorting algorithms: (a) Insertion sort (b) Merge sort (c) Quick sort (d)Heap sort.	
15	Write programs for implementation of graph traversals by applying: (a) BFS (b) DFS.	

Subject In-charge: Prof. Khalid Alfatmi	**APPROVED BY:** Dr. Makarand Shahade HOD, Department of Computer Engineering

Rubrics for Assessment

Assignment/Experiment :

Date of Performance: **Date of submission :**

Marks Split Up to	Maximum Marks	Marks Obtained
Performance/ conduction	3	
Report Writing	3	
Attendance	2	
Viva/Oral	2	
Total Marks	10	
Signature of Subject Teacher		

Course Objectives:

The objective of this lab is to teach students various data structures and to explain them algorithms for performing various operations on these data structures. This lab complements the data structures course. Students will gain practical knowledge by writing and executing programs in C using various data structures such as arrays, linked lists, stacks, queues, trees, graphs and search trees.

Course Outcomes:

At the end of this course Students will be able to:

CO1: To implement various concepts in stacks and Evaluate polish notation for given expression.

CO2: To implement concepts in queue such as circular queue as well as dequeue using array.

CO3: To design a stack using queues and perform basic operations in linear and constant time. Design a queue using stacks and perform dequeue operations in linear as well as in constant

CO4: To implement data structures as single and double linked list. Design stack using link list and perform stack operations with time complexity O (1).

CO5: To illustrate and implement concepts in trees and graphs and Construct Search trees.

CO6: To implement concepts in hashing and different sorting algorithms.

Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

DHULE (M.S.)

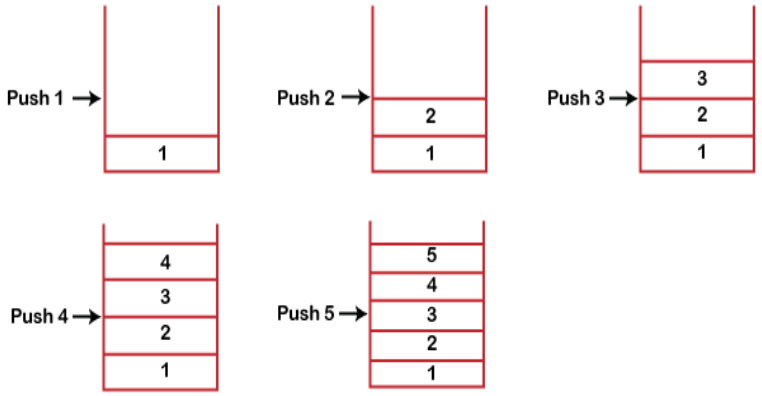
DEPARMENT OF COMPUTER ENGINEERING

Subject : Data Structure Lab **Subject Code :** BTCOL306

Class: S. Y. Comp. Engg. **Expt. No. :** 01

Title : Write a program to implement stack using arrays.

Problem Staement :	The main aim of this lab is to understand various concepts in stack and implement various operations associated with stack such as a. Push b. Pop c. Stack Overflow
Software Required :	Code Blocks
Theory :	A Stack is a linear data structure that follows the LIFO (Last-In-First-Out) principle. Stack has one end, whereas the Queue has two ends (front and rear). It contains only one pointer top pointer pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack. Some key points related to stack • It is called as stack because it behaves like a real-world stack, piles of books, etc. • A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size. • It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO. **Working of Stack** Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5. Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full. Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top.



When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements

PUSH operation

The steps involved in the PUSH operation is given below:

1. Before inserting an element in a stack, we check whether the stack is full.
2. If we try to insert the element in a stack, and the stack is full, then the overflow condition occurs.
3. When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
4. When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., top=top+1, and the element will be placed at the new position of the top.
5. The elements will be inserted until we reach the max size of the stack.

POP operation

1. The steps involved in the POP operation is given below:
2. Before deleting the element from the stack, we check whether the stack is empty.
3. If we try to delete the element from the empty stack, then the underflow condition occurs.
4. If the stack is not empty, we first access the element which is pointed by the top
5. Once the pop operation is performed, the top is decremented by 1, top=top-1

Programming Language :	**C** programming
Code:	Print Outs to be attached with executable code in C.
Output:	Screenshot Print outs for Executed output with all operations (Screenshot for Push, Pop Overflow and Underflow conditions.)

Code:

```
#include<stdio.h>
#include<conio.h>
int stk[100];
int n,top=-1;
int ch,data,i;
void push();
void pop();
void display();
int main(){
    printf("Enter the size of stack");
    scanf("%d",&n);
    do{
        printf("\nSelect the operation to be performed on
stack\n1.Push\n2.Pop\n3.Display");
    scanf("%d",&ch);
    switch(ch){
        case 1:push();
               break;
        case 2:pop();
               break;
        case 3:display();
               break;
        default:printf("\nInvalid input");
    }
    printf("\nDo you want to continue press 1");
    scanf("%d",&ch);
}while(ch==1);
}
void push(){
    if(top==n)
        printf("\n Stack Overflow");
    else if(top==-1){
        printf("\nEnter data:");
        scanf("%d",&data);
        top=0;
        stk[top]=data;
        top++;
    }
    else{
        printf("\nEnter data:");
        scanf("%d",&data);
        stk[top]=data;
        top++;
    }
}
void pop(){
    if(top==-1)
        printf("\nStack Underflow");
```

```
    else{
        data=stk[top];
        top--;
        printf("\nRemoved data is %d",data);
    }
}
void display(){
    printf("\nStack entries are");
    for(i=top;i>-1;i--)
    printf("\n%d",stk[i]);
}
```

Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

DHULE (M.S.)

DEPARMENT OF COMPUTER ENGINEERING

Subject : Data Structure Lab **Subject Code :** BTCOL306

Class: S. Y. Comp. Engg. **Expt. No. :** 02

Title : Write a program to evaluate a given postfix expression using stacks

Problem Staement :	Write a program to evaluate a given postfix expression using stacks
Software Required :	Code Blocks
Theory :	The way to write arithmetic expression is known as a notation. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are – • Infix • Prefix • Postfix ***Postfix Notation*** This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfixed** to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**. **Example**

Expression No	Infix Notation	Postfix Notation
1	a + b	a b +
2	(a + b) * c	a b + c *
3	a * (b + c)	a b c + *
4	a / b + c / d	a b / c d / +
5	(a + b) * (c + d)	a b + c d + *
6	((a + b) * c) - d	a b + c * d -

Parsing Expression

As we have discussed, it is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

	Precedence When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example – $a + b * c \rightarrow a + (b * c)$ As multiplication operation has precedence over addition, b * c will be evaluated first. A table of operator precedence is provided late
Algorithm	**Step 1:** If a character is an operand push it to Stack **Step 2:** If the character is an operator Pop two elements from the Stack. Operate on these elements according to the operator, and push the result back to the Stack **Step 3:** Step 1 and 2 will be repeated until the end has reached. **Step 4:** The Result is stored at the top of the Stack, return it **Step 5:** End
Programming Language :	**C** programming
Code:	Print Outs to be attached with executable code in C.
Output:	Screenshot Print outs for Executed output with all operations

Code

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

#define MAXSTACK 100 /* for max size of stack */
#define POSTFIXSIZE 100 /* define max number of charcters in postfix expression */

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) /* ch is an operator */
         {
         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("There are only four operators(*, /, +, -) in an expression and operand is single digit");
   printf(" \nEnter postfix expression,\npress right parenthesis ')' for end expression : ");
   for (i = 0; i <= POSTFIXSIZE - 1; i++) {
      scanf("%c", &postfix[i]);
   if (postfix[i] == ')') /* is there any way to eliminate this if */
      {
         break;
      } /* and break statement */
   }
   EvalPostfix(postfix);
   return 0;
}
```

Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

DHULE (M.S.)

DEPARMENT OF COMPUTER ENGINEERING

Subject : Data Structure Lab **Subject Code :** BTCOL306

Class: S. Y. Comp. Engg. **Expt. No. :** 03

Title : Write a program to convert a given infix expression to postfix form using stacks

Problem Staement :	Write a program to convert a given infix expression to postfix form using stacks.
Software Required :	Code Blocks
Theory :	A postfix expression is a collection of operators and operands in which the operator is placed after the operands. That means, in a postfix expression the operator follows the operands. Postfix Expression has following general structure... **Operand1 Operand2 Operator** **Example of Infix to Postfix Conversion** **Infix Expression:** A+(B*C+D)/E

Input Token	Stack	Postfix Expression	Action
A		A	Add A into expression string
+	+	A	Push '+' into stack
(	+(	A	Push (into stack
B	+(	AB	Add B into expression string
*	+(*	AB	Push '*' into stack
C	+(*	ABC	Add C into expression string
+	+(+	ABC*	'+' operator has less precedence than '*', so pop * and add to expression string
D	+(+	ABC*D	Add D into expression string

	)	+	ABC*D+	) has come so pop + and add it to expression string
	/	+/	ABC*D+	/ has higher precedence than + so push / into stack
	E	+/	ABC*D+E/+	Add E into expression string and pop all operators one by one from stack and add it to expression string

Postfix Notation

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfixed** to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.

Parsing Expression

As we have discussed, it is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

Precedence

When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example –

$a + b * c \rightarrow a + (b * c)$

As multiplication operation has precedence over addition, b * c will be evaluated first. A table of operator precedence is provided late

Algorithm

Algorithm to convert infix to postfix

Iterate the given expression from left to right, one character at a time

Step 1: If the scanned character is an operand, put it into postfix expression.

Step 2: If the scanned character is an operator and operator's stack is empty, push operator into operators' stack.

Step 3: If the operator's stack is not empty, there may be following possibilities.
If the precedence of scanned operator is greater than the top most operator of operator's stack, push this operator into operator 's stack.
If the precedence of scanned operator is less than the top most operator of operator's stack, pop the operators from operator's stack until we find a low precedence operator than the scanned character.
If the precedence of scanned operator is equal, then check the associativity of the operator. If associativity left to right, then pop the operators from stack until we find a low precedence operator. If associativity right to left, then simply put into stack.
If the scanned character is opening round bracket (**'('**), push it into operator's stack.
If the scanned character is closing round bracket (**')'**), pop out operators from operator's stack until we find an opening bracket (**'('**).

	Repeat Step 1,2 and 3 till expression has character **Step 4:** Now pop out all the remaining operators from the operator's stack and push into postfix expression. **Step 5:** Exit
Programming Language :	**C** programming
Code:	Print Outs to be attached with executable code in C.
Output:	Screenshot Print outs for Executed output with all operations

Code

```
#include<stdio.h>
#include<stdlib.h>      /* for exit() */
#include<ctype.h>     /* for isdigit(char ) */
#include<string.h>
#define SIZE 100

char stack[SIZE];
int top = -1;
void push(char item)
{
if(top >= SIZE-1)
printf("\nStack Overflow.");
else
{
top = top+1;
stack[top] = item;
}
}

char pop()
{
char item ;
if(top <0)
{
printf("stack under flow: invalid infix expression");
getchar();
exit(1);
}
else
{
item = stack[top];
top = top-1;
return(item);
}
}

int is_operator(char symbol)
{
if(symbol == '^' || symbol == '*' || symbol == '/' || symbol == '+' || symbol =='-')
return 1;
else
return 0;
}
int precedence(char symbol)
{
if(symbol == '^')/* exponent operator, highest precedence*/
return(3);
else if(symbol == '*' || symbol == '/')
return(2);
```

```
else if(symbol == '+' || symbol == '-')          /* lowest precedence */
return(1);
else
return(0);
}

void InfixToPostfix(char infix_exp[], char postfix_exp[])
{
int i, j;
char item;
char x;

push('(');                              /* push '(' onto stack */
strcat(infix_exp,")");                  /* add ')' to infix expression */

i=0;
j=0;
item=infix_exp[i];         /* initialize before loop*/

while(item != '\0')        /* run loop till end of infix expression */
{
if(item == '(')
{
push(item);
}
else if( isdigit(item) || isalpha(item))
{
postfix_exp[j] = item;              /* add operand symbol to postfix expr */
j++;
}
else if(is_operator(item) == 1)        /* means symbol is operator */
{
x=pop();
while(is_operator(x) == 1 && precedence(x)>= precedence(item))
{
postfix_exp[j] = x;/* so pop all higher precendence operator and */
j++;
x = pop();                       /* add them to postfix expresion */
}
push(x);
push(item);                /* push current oprerator symbol onto stack */
}
else if(item == ')')         /* if current symbol is ')' then */
{
x = pop();                 /* pop and keep popping until */
while(x != '(')              /* '(' encounterd */
{
postfix_exp[j] = x;
j++;
x = pop();
```

```
}
}
else
{ /* if current symbol is neither operand not '(' nor ')' and nor
operator */
printf("\nInvalid infix Expression.\n");        /* the it is illegeal  symbol */
getchar();
exit(1);
}
i++;
item = infix_exp[i]; /* go to next symbol of infix expression */
} /* while loop ends here */
if(top>0)
{
printf("\nInvalid infix Expression.\n");        /* the it is illegeal  symbol */
getchar();
exit(1);
}
if(top>0)
{
printf("\nInvalid infix Expression.\n");        /* the it is illegeal  symbol */
getchar();
exit(1);
}


postfix_exp[j] = '\0'; /* add sentinel else puts() fucntion */
}

int main()
{
char infix[SIZE], postfix[SIZE];         /* declare infix string and postfix string */
printf("The infix expression contains single letter variables and single digit constants);
printf("\nEnter Infix expression : ");
gets(infix);
          InfixToPostfix(infix,postfix);                  /* call to convert */
printf("Postfix Expression: ");
puts(postfix);                    /* print postfix expression */

return 0;
}
```

Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

DHULE (M.S.)

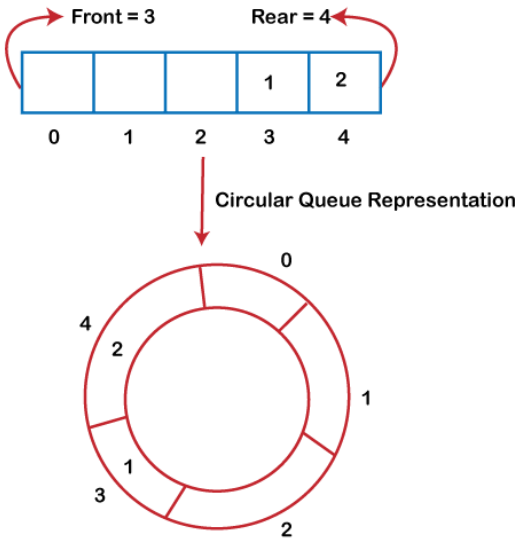
DEPARMENT OF COMPUTER ENGINEERING

Subject : Data Structure Lab **Subject Code :** BTCOL306

Class: S. Y. Comp. Engg. **Expt. No. :** 04

Title : Write a program to implement circular queue using arrays.

Problem Staement :	Write a program to implement circular queue using arrays.
Software Required :	Code Blocks
Theory :	Circular queue Why was the concept of the circular queue introduced? There was one limitation in the array implementation of Queue. If the rear reaches to the end position of the Queue, then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome such limitations, the concept of the circular queue was introduced. **Circular Queue** As we can see in the above image, the rear is at the last position of the Queue and front is pointing somewhere rather than the 0th position. In the above array, there are only two elements and other three positions are empty. The rear is at the last position of the Queue; if we try to insert the element then it will show that there are no empty spaces in the Queue. There is one solution to avoid such wastage of memory space by shifting both the elements at the left and adjust the front and rear end accordingly. It is not a practically good approach because shifting all the elements will consume lots of time. The efficient approach to avoid the wastage of the memory is to use the circular queue data structure.



What is a Circular Queue?

A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a Ring Buffer.

Operations on Circular Queue

The following are the operations that can be performed on a circular queue:
Front: It is used to get the front element from the Queue.
Rear: It is used to get the rear element from the Queue.
enQueue(value): This function is used to insert the new value in the Queue. The new element is always inserted from the rear end.
deQueue(): This function deletes an element from the Queue. The deletion in a Queue always takes place from the front end.
Applications of Circular Queue
The circular Queue can be used in the following scenarios:

Memory management: The circular queue provides memory management. As we have already seen that in linear queue, the memory is not managed very efficiently. But in case of a circular queue, the memory is managed efficiently by placing the elements in a location which is unused.

CPU Scheduling: The operating system also uses the circular queue to insert the processes and then execute them.

Traffic system: In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every jinterval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.

Algorithm

```
Insert CircularQueue ( )

If (FRONT == 1 and REAR == N) or (FRONT == REAR + 1) Then
    Print: Overflow
```

	Else
	If (REAR == 0) Then [Check if QUEUE is empty]
	(a) Set FRONT = 1
	(b) Set REAR = 1
	Else If (REAR == N)
	Then [If REAR reaches end if QUEUE]
	Set REAR = 1
	Else
	Set REAR = REAR + 1 [Increment REAR by 1]
	[End of Step 4 If]
	Set QUEUE[REAR] = ITEM
	Print: ITEM inserted
	[End of Step 1 If]
	Exit

Programming Language :	**C** programming
Code:	Print Outs to be attached with executable code in C.
Output:	Screenshot Print outs for Executed output with all operations

CODE:

```
# include<stdio.h>
# define MAX 5

int cqueue_arr[MAX];
int front = -1;
int rear = -1;

void insert(int item)
{
        if((front == 0 && rear == MAX-1) || (front == rear+1))
        {
                printf("Queue Overflow \n");
                return;
        }
        if (front == -1)  /*If queue is empty */
        {
                front = 0;
                rear = 0;
        }
        else
        {
                if(rear == MAX-1)      /*rear is at last position of queue */
                        rear = 0;
                else
                        rear = rear+1;
        }
        cqueue_arr[rear] = item ;
}

void del()
{
        if (front == -1)
        {
                printf("Queue Underflow\n");
                return ;
        }
        printf("Element deleted from queue is : %d\n",cqueue_arr[front]);
        if(front == rear) /* queue has only one element */
        {
                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_arr[front_pos]);
                        front_pos++;
                }
        else
        {
                while(front_pos <= MAX-1)
                {
                        printf("%d ",cqueue_arr[front_pos]);
                        front_pos++;
                }
                front_pos = 0;
                while(front_pos <= rear_pos)
                {
                        printf("%d ",cqueue_arr[front_pos]);
                        front_pos++;
                }
        }
        printf("\n");
}
int main()
{
        int choice,item;
        do
        {
                printf("1.Insert\n");
                printf("2.Delete\n");
                printf("3.Display\n");
                printf("4.Quit\n");

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

                switch(choice)
                {
                        case 1 :printf("Input the 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;
}
```

Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

DHULE (M.S.)

DEPARMENT OF COMPUTER ENGINEERING

Subject : Data Structure Lab **Subject Code :** BTCOL306

Class: S. Y. Comp. Engg. **Expt. No. :** 05

Title : Write a program to implement double ended queue (dequeue) using arrays.

Problem Staement :	Write a program to implement double ended queue (dequeue) using arrays
Software Required :	Code Blocks
Theory :	A queue is a data structure in which whatever comes first will go out first, and it follows the FIFO (First-In-First-Out) policy. Insertion in the queue is done from one end known as the rear end or the tail, whereas the deletion is done from another end known as the front end or the head of the queue. The deque stands for Double Ended Queue. Deque is a linear data structure where the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue. Though the insertion and deletion in a deque can be performed on both ends, it does not follow the FIFO rule. The representation of a deque is given as follows –

Representation of deque

Operations performed on deque

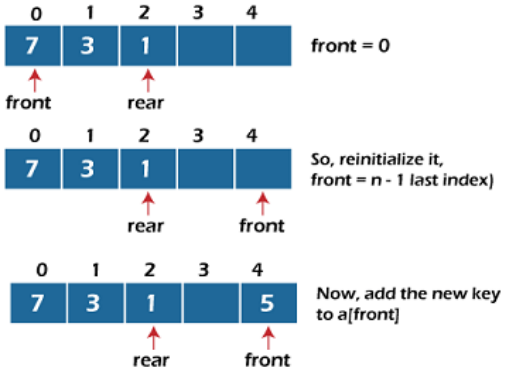
There are the following operations that can be applied on a deque -

1. Insertion at front
2. Insertion at rear
3. Deletion at front
4. Deletion at rear

Insertion at the front end

In this operation, the element is inserted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is full or not. If the queue is not full, then the element can be inserted from the front end by using the below conditions –

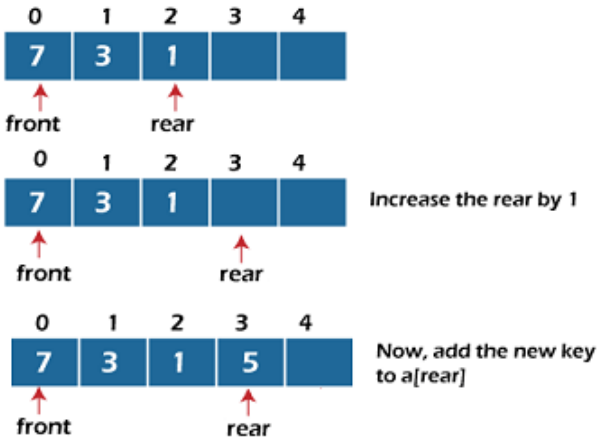
1. If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
2. Otherwise, check the position of the front if the front is less than 1 (front < 1), then reinitialize it by **front = n - 1**, i.e., the last index of the array.



Insertion at the rear end

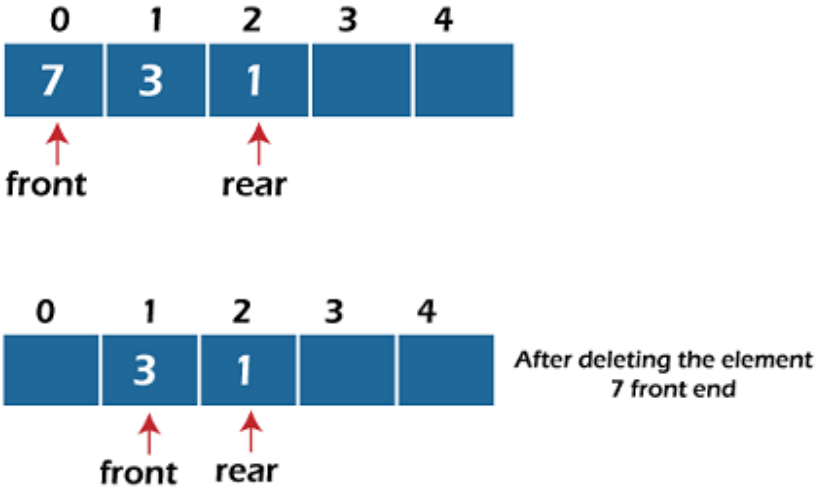
In this operation, the element is inserted from the rear end of the queue. Before implementing the operation, we first have to check again whether the queue is full or not. If the queue is not full, then the element can be inserted from the rear end by using the below conditions -

- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, increment the rear by 1. If the rear is at last index (or size - 1), then instead of increasing it by 1, we have to make it equal to 0.



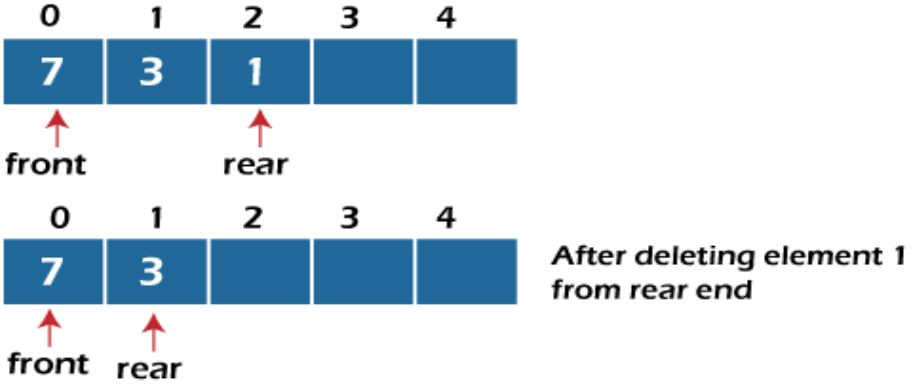
In this operation, the element is deleted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.
If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

- If the deque has only one element, set rear = -1 and front = -1.
- Else if front is at end (that means front = size - 1), set front = 0.
- Else increment the front by 1, (i.e., front = front + 1).



Deletion at the rear end

In this operation, the element is deleted from the rear end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.
If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion.
If the deque has only one element, set rear = -1 and front = -1.
If rear = 0 (rear is at front), then set rear = n - 1.
Else, decrement the rear by 1 (or, rear = rear -1).



Algorithm

Algorithm for Insertion at rear end

```
Step-1: [Check for overflow]
        if(rear==MAX)
                Print("Queue is Overflow");
                return;
Step-2:  [Insert Element]
        else
                rear=rear+1;
                q[rear]=no;
        [Set rear and front pointer]
        if rear=0
                rear=1;
        if front=0
                front=1;
Step-3: return
```

Algorithm for Insertion at front end

```
Step-1 :  [Check for the front position]
        if(front<=1)
                Print("Cannot add item at the front");
                return;
Step-2 :  [Insert at front]
        else
                front=front-1;
                q[front]=no;
Step-3  : Return
```

Algorithm for Deletion from front end

```
Step-1 [ Check for front pointer]
        if front=0
                print(" Queue is Underflow");
                return;
Step-2 [Perform deletion]
        else
                no=q[front];
                print("Deleted element is",no);
        [Set front and rear pointer]
        if front=rear
                front=0;
                rear=0;
        else
```

	front=front+1;
	Step-3 : Return
	
	Algorithm for Deletion from rear end
	Step-1 : [Check for the rear pointer]
	if rear=0
	print("Cannot delete value at rear end");
	return;
	Step-2: [perform deletion]
	else
	no=q[rear];
	[Check for the front and rear pointer]
	if front= rear
	front=0;
	rear=0;
	else
	rear=rear-1;
	print("Deleted element is",no);
	Step-3 : Return

Programming Language :	**C** programming
Code:	Print Outs to be attached with executable code in C.
Output:	Screenshot Print outs for Executed output with all operations

CODE:

```
# include<stdio.h>
# define MAX 5

int deque_arr[MAX];
int left = -1;
int right = -1;

void insert_right()
{
        int added_item;
        if((left == 0 && right == MAX-1) || (left == right+1))
        {       printf("Queue Overflow\n");
                return;}
        if (left == -1)  /* if queue is initially empty */
        {       left = 0;
                right = 0;}
        else
        if(right == MAX-1)  /*right is at last position of queue */
                right = 0;
        else
                right = right+1;
        printf("Input the element for adding in queue : ");
        scanf("%d", &added_item);
        deque_arr[right] = added_item ;
}

void insert_left()
{       int added_item;
        if((left == 0 && right == MAX-1) || (left == right+1))
        {       printf("Queue Overflow \n");
                return;  }
        if (left == -1)/*If queue is initially empty*/
        {       left = 0;
                right = 0;        }
        else
        if(left== 0)
                left=MAX-1;
        else
                left=left-1;
        printf("Input the element for adding in queue : ");
        scanf("%d", &added_item);
        deque_arr[left] = added_item ;          }

void delete_left()
{       if (left == -1)
        {       printf("Queue Underflow\n");
                return ;}
        printf("Element deleted from queue is : %d\n",deque_arr[left]);
```

```
        if(left == right) /*Queue has only one element */
        {       left = -1;
                right=-1;       }
        else
                if(left == MAX-1)
                        left = 0;
                else
                        left = left+1;
}

void delete_right()
{if (left == -1)
        {printf("Queue Underflow\n");
                return ; }
        printf("Element deleted from queue is : %d\n",deque_arr[right]);
        if(left == right) /*queue has only one element*/
        {       left = -1;
                right=-1;       }
        else
                if(right == 0)
                        right=MAX-1;
                else
                        right=right-1;  }
void display_queue()
{       int front_pos = left,rear_pos = right;
        if(left == -1)
        {       printf("Queue is empty\n");
                return;  }
        printf("Queue elements :\n");
        if( front_pos <= rear_pos )
        {       while(front_pos <= rear_pos)
                {       printf("%d ",deque_arr[front_pos]);
                        front_pos++;   }        }
        else
        {       while(front_pos <= MAX-1)
                {       printf("%d ",deque_arr[front_pos]);
                        front_pos++;   }
                front_pos = 0;
                while(front_pos <= rear_pos)
                {       printf("%d ",deque_arr[front_pos]);
                        front_pos++;
                }
        }
        printf("\n");
}
void input_que()
{       int choice;
        do
        {       printf("1.Insert at right\n");
                printf("2.Delete from left\n");
```

```
                printf("3.Delete from right\n");
                printf("4.Display\n");
                printf("5.Quit\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);

                switch(choice)
                {       case 1:insert_right();
                        break;
                 case 2:delete_left();
                        break;
                 case 3:delete_right();
                        break;
                 case 4:display_queue();
                        break;
                 case 5:break;
                 default:printf("Wrong choice\n");
                }
        }while(choice!=5);
}
void output_que()
{       int choice;
        do
        {       printf("1.Insert at right\n");
                printf("2.Insert at left\n");
                printf("3.Delete from left\n");
                printf("4.Display\n");
                printf("5.Quit\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);
                switch(choice)
                {
                 case 1:insert_right();
                        break;
                 case 2:insert_left();
                        break;
                 case 3:delete_left();
                        break;
                 case 4:display_queue();
                        break;
                 case 5:break;
                 default:printf("Wrong choice\n");
                }
        }while(choice!=5);
}
main()
{       int choice;
        printf("1.Input restricted dequeue\n");
        printf("2.Output restricted dequeue\n");
        printf("Enter your choice : ");
```

```
        scanf("%d",&choice);
        switch(choice)
        {
         case 1 :input_que();
                 break;
         case 2:output_que();
                 break;
         default:printf("Wrong choice\n");
        }
}
```

Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

DHULE (M.S.)

DEPARMENT OF COMPUTER ENGINEERING

Subject : Data Structure Lab **Subject Code :** BTCOL306

Class: S. Y. Comp. Engg. **Expt. No. :** 06

Title : . Write a program to implement a stack using two queues such that the push operation runs in constant time and the pop operation runs in linear time.

Problem Staement :	Write a program to implement a stack using two queues such that the push operation runs in constant time and the pop operation runs in linear time.
Software Required :	Code Blocks
Theory :	
Algorithm	
Programming Language :	**C** programming
Code:	Print Outs to be attached with executable code in C.
Output:	Screenshot Print outs for Executed output with all operations

CODE :

```
#include <stdio.h>
#include <stdlib.h>
#define QUEUE_EMPTY_MAGIC 0xdeadbeef
typedef struct _queue_t {
    int *arr;
    int rear, front, count, max;
} queue_t;


queue_t *queue_allocate(int n);
void queue_insert(queue_t * q, int v);
int queue_remove(queue_t * q);
int queue_count(queue_t * q);
int queue_is_empty(queue_t * q);

void stack_push(queue_t * q, int v) {
    queue_insert(q, v);
}

int stack_pop(queue_t * q) {
    int i, n = queue_count(q);
    int removed_element;
    for (i = 0; i < (n - 1); i++) {
        removed_element = queue_remove(q);
        queue_insert(q, removed_element);
        /* same as below */
        //queue_insert (q, queue_remove (q))
    }
    removed_element = queue_remove(q);
    return removed_element;
}

int stack_is_empty(queue_t * q) {
    return queue_is_empty(q);
}

int stack_count(queue_t * q) {
    return queue_count(q);
}

int queue_count(queue_t * q) {
    return q->count;
}

queue_t *
queue_allocate(int n) {
    queue_t *queue;
```

```
    queue = malloc(sizeof(queue_t));
    if (queue == NULL)
        return NULL;
    queue->max = n;
    queue->arr = malloc(sizeof(int) * n);
    queue->rear = n - 1;
    queue->front = n - 1;
    return queue;
}

void queue_insert(queue_t * q, int v) {
    if (q->count == q->max)
        return;

    q->rear = (q->rear + 1) % q->max;
    q->arr[q->rear] = v;
    q->count++;
}

int queue_remove(queue_t * q) {
    int retval;
    if (q->count == 0)
        return QUEUE_EMPTY_MAGIC;
    q->front = (q->front + 1) % q->max;
    retval = q->arr[q->front];
    q->count--;
    return retval;
}

int queue_is_empty(queue_t * q) {
    return (q->count == 0);
}

void queue_display(queue_t * q) {
    int i = (q->front + 1) % q->max, elements = queue_count(q);

    while (elements--) {
        printf("[%d], ", q->arr[i]);
        i = (i >= q->max) ? 0 : (i + 1);
    }
}

#define MAX 128
int main(void) {
    queue_t *q;
    int x, select;
    /* Static allocation */
    q = queue_allocate(MAX);
```

```
    do {
        printf("\n[1] Push\n[2] Pop\n[0] Exit");
        printf("\nChoice: ");
        scanf(" %d", &select);

        switch (select) {
        case 1:
            printf("\nEnter value to Push:");
            scanf(" %d", &x);
            stack_push(q, x); /* Pushing */
            printf("\n\n_____________________________\nCurrent Queue:\n");
            queue_display(q);
            printf("\n\nPushed Value: %d", x);
            printf("\n_____________________________\n");
            break;

        case 2:
            x = stack_pop(q); /* Popping */
            printf("\n\n\n\n_____________________________\nCurrent
Queue:\n");
            queue_display(q);
            if (x == QUEUE_EMPTY_MAGIC)
                printf("\n\nNo values removed");
            else
                printf("\n\nPopped Value: %d", x);

            printf("\n_____________________________\n");
            break;

        case 0:
            printf("\nQutting.\n");
            return 0;

        default:
            printf("\nQutting.\n");
            return 0;
        }
    } while (1);

    return 0;
}
```

Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

DHULE (M.S.)

DEPARMENT OF COMPUTER ENGINEERING

Subject : Data Structure Lab **Subject Code :** BTCOL306

Class: S. Y. Comp. Engg. **Expt. No. :** 07

Title : Write a program to implement a stack using two queues such that the push operation runs in linear time and the pop operation runs in constant time.

Problem Staement :	Write a program to implement a stack using two queues such that the push operation runs in linear time and the pop operation runs in constant time.
Software Required :	Code Blocks
Theory :	
Algorithm	
Programming Language :	**C** programming
Code:	Print Outs to be attached with executable code in C.
Output:	Screenshot Print outs for Executed output with all operations

Code :

```
/* C program to implement queues using two stacks */
#include <stdio.h>
#include <stdlib.h>
struct node
{
    int data;
    struct node *next;
};
void push(struct node** top, int data);
int pop(struct node** top);
struct queue
{
    struct node *stack1;
    struct node *stack2;
```

```
};
void enqueue(struct queue *q, int x)
{
    push(&q->stack1, x);
}
void dequeue(struct queue *q)
{
    int x;
    if (q->stack1 == NULL && q->stack2 == NULL) {
        printf("queue is empty");
        return;
    }
    if (q->stack2 == NULL) {
        while (q->stack1 != NULL) {
        x = pop(&q->stack1);
        push(&q->stack2, x);
        }
    }
    x = pop(&q->stack2);
    printf("%d\n", x);
}
void push(struct node** top, int data)
{
    struct node* newnode = (struct node*) malloc(sizeof(struct node));
        if (newnode == NULL) {
            printf("Stack overflow \n");
            return;
        }
    newnode->data = data;
    newnode->next = (*top);
    (*top) = newnode;
}
int pop(struct node** top)
{
    int buff;
    struct node *t;
    if (*top == NULL) {
        printf("Stack underflow \n");
        return;
    }
    else {
        t = *top;
        buff = t->data;
        *top = t->next;
        free(t);
        return buff;
    }
}
void display(struct node *top1,struct node *top2)
```

```
{
    while (top1 != NULL) {
        printf("%d\n", top1->data);
        top1 = top1->next;
    }
    while (top2 != NULL) {
        printf("%d\n", top2->data);
        top2 = top2->next;
    }
}
int main()
{
    struct queue *q = (struct queue*)malloc(sizeof(struct queue));
    int f = 0, a;
    char ch = 'y';
    q->stack1 = NULL;
    q->stack2 = NULL;
    while (ch == 'y'||ch == 'Y') {
        printf("enter ur choice\n1.add to queue\n2.remove
               from queue\n3.display\n4.exit\n");
        scanf("%d", &f);
        switch(f) {
            case 1 : printf("enter the element to be added to queue\n");
                     scanf("%d", &a);
                     enqueue(q, a);
                     break;
            case 2 : dequeue(q);
                     break;
            case 3 : display(q->stack1, q->stack2);
                     break;
            case 4 : exit(1);
                     break;
            default : printf("invalid\n");
                      break;
        }
    }
}


           case 6:deleteNode(&head, head->next->next->next->next->next);
                 break;
           case 8:exit(1);
                 break;
           default:printf("Incorrect Choice\n");
           }
      }
}
```

Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

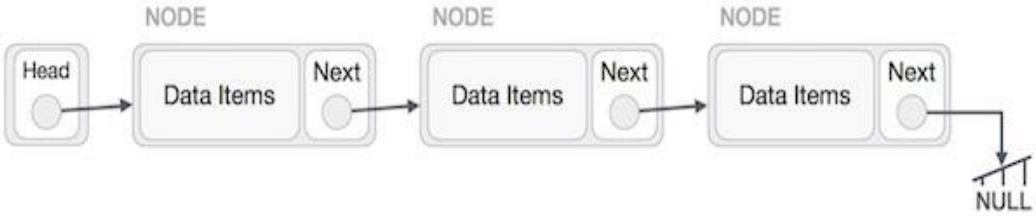
DHULE (M.S.)

DEPARMENT OF COMPUTER ENGINEERING

Subject : Data Structure Lab **Subject Code :** BTCOL306

Class: S. Y. Comp. Engg. **Expt. No. :** 09

Title : Write programs to implement the following data structures:
(a) Single linked list
(b) Double linked list.

Problem Staement :	Write programs to implement the following data structures: (a) Single linked list (b) Double linked list.
Software Required :	Code Blocks
Theory :	A linked list is a sequence of data structures, which are connected together via links. Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List. • Link − Each link of a linked list can store a data called an element. • Next − Each link of a linked list contains a link to the next link called Next. • LinkedList − A Linked List contains the connection link to the first link called First. **Linked List Representation** Linked list can be visualized as a chain of nodes, where every node points to the next node.  As per the above illustration, following are the important points to be considered. • Linked List contains a link element called first. • Each link carries a data field(s) and a link field called next. • Each link is linked with its next link using its next link. Last link carries a link as null to mark the end of the list.

Types of Linked List

Following are the various types of linked list.

1. Simple Linked List − Item navigation is forward only.
2. Doubly Linked List − Items can be navigated forward and backward.
3. Circular Linked List − Last item contains link of the first element as next and the first element has a link to the last element as previous.

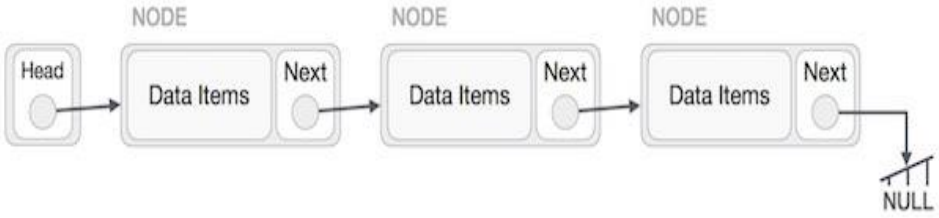
Basic Operations

Following are the basic operations supported by a list.

1. Insertion − Adds an element at the beginning of the list.
2. Deletion − Deletes an element at the beginning of the list.
3. Display − Displays the complete list.
4. Search − Searches an element using the given key.
5. Delete − Deletes an element using the given key.

Insertion Operation

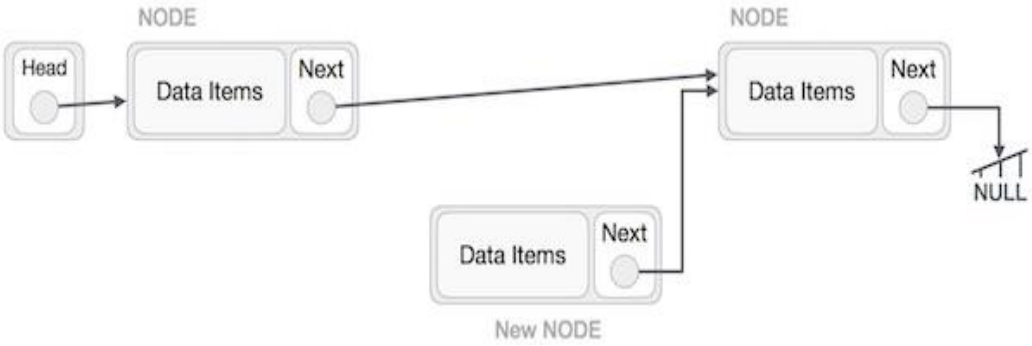
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



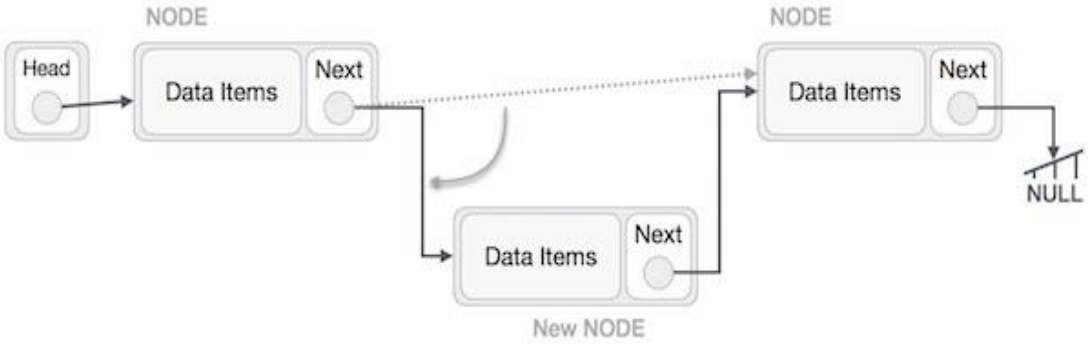
Imagine that we are inserting a node B (NewNode), between A (LeftNode) and C (RightNode). Then point B.next to C −

NewNode.next −> RightNode;

It should look like this −

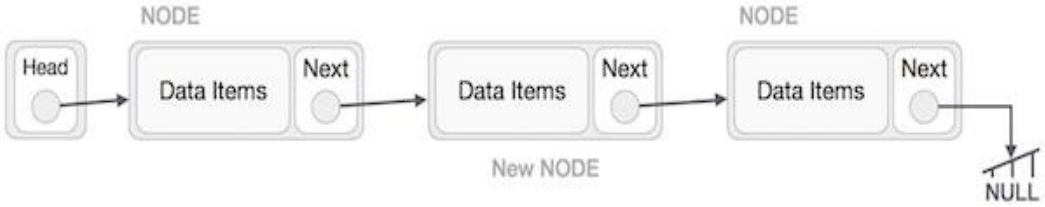


Now, the next node at the left should point to the new node.



LeftNode.next −> NewNode;

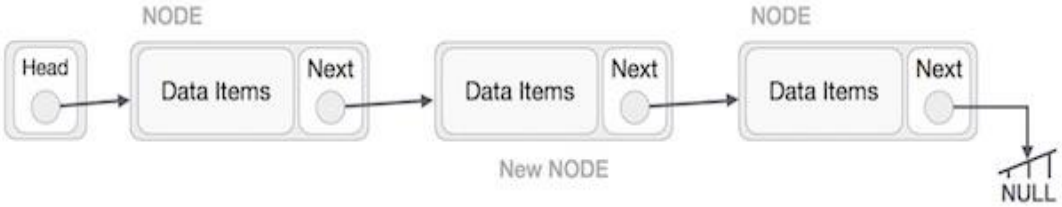
This will put the new node in the middle of the two. The new list should look like this −



Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

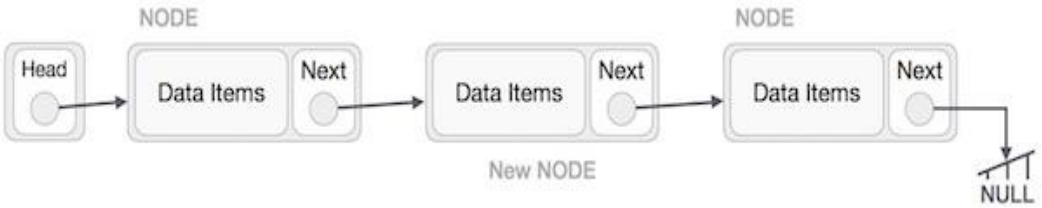
Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



The left (previous) node of the target node now should point to the next node of the target node –

LeftNode.next −> TargetNode.next;



This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

TargetNode.next −> NULL;

We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

Algorithm	
Programming Language :	**C** programming
Code:	Print Outs to be attached with executable code in C.
Output:	Screenshot Print outs for Executed output with all operations

CODE: SINGLE LINK LIST

```
#include <stdio.h>
#include <stdlib.h>
struct Node // Create a node
{
  int data;
  struct Node* next;
};

void insertAtBeginning(struct Node** head_ref, int new_data) // at beginning
 {
  struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
  new_node->data = new_data;   // insert the data
  new_node->next = (*head_ref);  // Move head to new node
  (*head_ref) = new_node;
}

void insertAfter(struct Node* prev_node, int new_data) // Insert a node after a node
{
  if (prev_node == NULL) {
  printf("the given previous node cannot be NULL");
  return;
  }

  struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
  new_node->data = new_data;
  new_node->next = prev_node->next;
  prev_node->next = new_node;
}

void insertAtEnd(struct Node** head_ref, int new_data) // Insert the the end
{
  struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
  struct Node* last = *head_ref; /* used in step 5*/
  new_node->data = new_data;
  new_node->next = NULL;

  if (*head_ref == NULL) {
  *head_ref = new_node;
  return;
  }

  while (last->next != NULL) last = last->next;

  last->next = new_node;
  return;
}

void deleteNode(struct Node** head_ref, int key) // Delete a node
{
  struct Node *temp = *head_ref, *prev;
  if (temp != NULL && temp->data == key) {
  *head_ref = temp->next;
  free(temp);
  return;
  }
  // Find the key to be deleted
  while (temp != NULL && temp->data != key) {
```

```
  prev = temp;
  temp = temp->next;
  }

  // If the key is not present
  if (temp == NULL) return;

  // Remove the node
  prev->next = temp->next;

  free(temp);
}

int searchNode(struct Node** head_ref, int key) // Search a node

 {
  struct Node* current = *head_ref;
  while (current != NULL) {
  if (current->data == key) return 1;
  current = current->next;
  }
  return 0;
}

void sortLinkedList(struct Node** head_ref)  // Sort the linked list
{
  struct Node *current = *head_ref, *index = NULL;
  int temp;

  if (head_ref == NULL) {
  return;
  } else {
  while (current != NULL) {
    index = current->next; // index points to the node next to current

    while (index != NULL) {
    if (current->data > index->data) {
      temp = current->data;
      current->data = index->data;
      index->data = temp;
    }
    index = index->next;
    }
    current = current->next;
  }
  }
}

void printList(struct Node* node) // Print the linked list
{
  while (node != NULL) {
  printf(" %d ", node->data);
  node = node->next;
  }
}

int main() {
  struct Node* head = NULL;
```

```
  insertAtEnd(&head, 1);
  insertAtBeginning(&head, 2);
  insertAtBeginning(&head, 3);
  insertAtEnd(&head, 4);
  insertAfter(head->next, 5);

  printf("Linked list: ");
  printList(head);

  printf("\nAfter deleting an element: ");
  deleteNode(&head, 3);
  printList(head);

  int item_to_find = 3;
  if (searchNode(&head, item_to_find)) {
  printf("\n%d is found", item_to_find);
  } else {
  printf("\n%d is not found", item_to_find);
  }

  sortLinkedList(&head);
  printf("\nSorted List: ");
  printList(head);
}
```

CODE: Double Linked List

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

// node creation
struct Node {
  int data;
  struct Node* next;
  struct Node* prev;
};

// insert node at the front
void insertFront(struct Node** head, int data) {  // allocate memory for newNode
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = data; // assign data to newNode
  newNode->next = (*head); // make newNode as a head
  newNode->prev = NULL; // assign null to prev

  // previous of head (now head is the second node) is newNode
  if ((*head) != NULL)
    (*head)->prev = newNode;
    (*head) = newNode; // head points to newNode
}

// insert a node after a specific node
void insertAfter(struct Node* prev_node, int data) {
  // check if previous node is null
  if (prev_node == NULL) {
    printf("previous node cannot be null");
    return;
  }

  // allocate memory for newNode
```

```
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = data;  // assign data to newNode
  newNode->next = prev_node->next; // set next of newNode to next of prev node
  prev_node->next = newNode; // set next of prev node to newNode
  newNode->prev = prev_node; // set prev of newNode to the previous node
  if (newNode->next != NULL) // set prev of newNode's next to newNode
    newNode->next->prev = newNode;
}

// insert a newNode at the end of the list
void insertEnd(struct Node** head, int data) {
  // allocate memory for node
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = data; // assign data to newNode
  newNode->next = NULL; // assign null to next of newNode
  struct Node* temp = *head; // store the head node temporarily (for later use)

  // if the linked list is empty, make the newNode as head node
  if (*head == NULL) {
    newNode->prev = NULL;
    *head = newNode;
    return;
  }

  // if the linked list is not empty, traverse to the end of the linked list
  while (temp->next != NULL)
    temp = temp->next;

  // now, the last node of the linked list is temp
  temp->next = newNode; // assign next of the last node (temp) to newNode
  newNode->prev = temp; // assign prev of newNode to temp
}

// delete a node from the doubly linked list
void deleteNode(struct Node** head, struct Node* del_node) {
  // if head or del is null, deletion is not possible
  if (*head == NULL || del_node == NULL)
    return;

  // if del_node is the head node, point the head pointer to the next of del_node
  if (*head == del_node)
    *head = del_node->next;

  // if del_node is not at the last node, point the prev of node next to del_node to the
previous of del_node
  if (del_node->next != NULL)
    del_node->next->prev = del_node->prev;

  // if del_node is not the first node, point the next of the previous node to the next
node of del_node
  if (del_node->prev != NULL)
    del_node->prev->next = del_node->next;
  free(del_node); // free the memory of del_node
}

// print the doubly linked list
void displayList(struct Node* node) {
  struct Node* last;
  while (node != NULL) {
```

```
    printf("%d->", node->data);
    last = node;
    node = node->next;
  }
  if (node == NULL)
    printf("NULL\n");
}

int main() {

  struct Node* head = NULL;  // initialize an empty node
  insertEnd(&head, 5);
  insertFront(&head, 1);
  insertFront(&head, 6);
  insertEnd(&head, 9);
  insertAfter(head, 11); // insert 11 after head
  insertAfter(head->next, 15); // insert 15 after the second node
  displayList(head);
  deleteNode(&head, head->next->next->next->next->next); // delete the last node
  displayList(head);
}
```

Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

DHULE (M.S.)

DEPARMENT OF COMPUTER ENGINEERING

Subject : Data Structure Lab **Subject Code :** BTCOL306

Class: S. Y. Comp. Engg. **Expt. No. :** 10

Title : Write a program to implement a stack using a linked list such that the push and pop operations of stack still take O(1)time

Problem Staement :	Write a program to implement a stack using a linked list such that the push and pop operations of stack still take O(1)time
Software Required :	Code Blocks
Theory :	
Algorithm	
Programming Language :	**C** programming
Code:	Print Outs to be attached with executable code in C.
Output:	Screenshot Print outs for Executed output with all operations

CODE:

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

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

struct node *head = NULL;

void push(int val)
{
    struct node *newNode = malloc(sizeof(struct node)); //create new node
    newNode->data = val;
    newNode->next = head; //make the new node points to the head node

    //make the new node as head node
    //so that head will always point the last inserted data
    head = newNode;
}

void pop()
{

    struct node *temp; //temp is used to free the head node
    if(head == NULL)
        printf("Stack is Empty\n");
    else
    {
        printf("Poped element = %d\n", head->data);
        temp = head; //backup the head node

        //make the head node points to the next node.
        //logically removing the node
        head = head->next;
        free(temp); //free the poped element's memory
    }
}

void printList()//print the linked list
{
    struct node *temp = head;

    //iterate the entire linked list and print the data
    while(temp != NULL)
    {
         printf("%d->", temp->data);
         temp = temp->next;
    }
    printf("NULL\n");
```

```
}
int data,ch;
int main()
{

    printf(“\nSelect operation 1.Push 2.Pop\n”);
    scanf(“%d”,&ch);
    switch(ch){
    case 1: printf(“\nEnter data to push on top of top:”);
            scanf(“%d”,&data);
            push(data);
            printf("\nLinked List\n");
            printList();
            break;
    case 2: pop();
            printf("\nAfter the pop, the new linked list\n");
            printList();
            break;
    default:printf(“\nInvalid operation”);
   }
    return 0;
}
```

Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

DHULE (M.S.)

DEPARMENT OF COMPUTER ENGINEERING

Subject : Data Structure Lab **Subject Code :** BTCOL306

Class: S. Y. Comp. Engg. **Expt. No. :** 10

Title : Write a program to implement a stack using a linked list such that the push and pop operations of stack still take O(1)time

Problem Staement :	Write a program to implement a stack using a linked list such that the push and pop operations of stack still take O(1)time
Software Required :	Code Blocks
Theory :	
Algorithm	
Programming Language :	**C** programming
Code:	Print Outs to be attached with executable code in C.
Output:	Screenshot Print outs for Executed output with all operations

CODE:

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

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

struct node *head = NULL;

void push(int val)
{
    struct node *newNode = malloc(sizeof(struct node)); //create new node
    newNode->data = val;
    newNode->next = head; //make the new node points to the head node

    //make the new node as head node
    //so that head will always point the last inserted data
    head = newNode;
}

void pop()
{

    struct node *temp; //temp is used to free the head node
    if(head == NULL)
        printf("Stack is Empty\n");
    else
    {
        printf("Poped element = %d\n", head->data);
        temp = head; //backup the head node

        //make the head node points to the next node.
        //logically removing the node
        head = head->next;
        free(temp); //free the poped element's memory
    }
}

void printList()//print the linked list
{
    struct node *temp = head;

    //iterate the entire linked list and print the data
    while(temp != NULL)
    {
```

```
        printf("%d->", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
int data,ch;
int main()
{

    printf(“\nSelect operation 1.Push 2.Pop\n”);
    scanf(“%d”,&ch);
    switch(ch){
    case 1: printf(“\nEnter data to push on top of top:”);
            scanf(“%d”,&data);
            push(data);
            printf("\nLinked List\n");
            printList();
            break;
    case 2: pop();
            printf("\nAfter the pop, the new linked list\n");
            printList();
            break;
    default:printf(“\nInvalid operation”);
   }
    return 0;
}
```