Class: D10A Seat No.____

Roll No: 01

# VIVEKANAND EDUCATION SOCIETY'S INSTITUTE OF TECHNOLOGY

Hashu Advani Memorial Complex, Collector's Colony, R C Marg, Chembur, Mumbai-400074


# CERTIFICATE

Certified that Mr./~~Miss~~ **Aamir Z Ansari** of **D10A** has satisfactorily completed a course of the necessary experiments/assignments in **Data Structures Lab** under my supervision in the Institute of technology in the year 2020 - 2021

Principal Head of Department

Lab In-charge Subject Teacher

**Vivekanand Education Society's Institute Of Technology**
**Department Of Information Technology**
**2020-2021**

**Name of the Course: Data Structures Lab**
**Year/Sem/Class:** S.E.(INFT) Sem III (D10A)
**Faculty In charge :** Mrs. Charusheela N. and Mrs. Dimple B.
**Email:** charusheela.nehete@ves.ac.in , dimple.bohra@ves.ac.in

| Sr. No. | Lab Experiments |
|---|---|
| 1 | Implementation of stack using array |
| 2 | Evaluation of Infix and Prefix Expression using Stack |
| 3 | Infix to Postfix Conversion using Stack. |
| 4 | Implementation of Linear Queue using array. |
| 5 | Implementation of circular queue using array. |
| 6 | Implementation of Singly linked lists |
| 7 | Linked List implementation of Stack |
| 8 | Implementation of Linear Queue using Singly LL. |
| 9 | Implementation of Singly Circular linked lists |
| 10 | Implementation of Josephus Problem using Circular LL |
| 11 | Implementation of DLL |
| 12 | Implementation of Circular Doubly linked lists |
| 13 | Implementation of various operations on binary search tree. |
| 14 | Implementation of AVL tree. |
| 15 | Implementation of BFS and DFS on a directed graph using adjacency matrix. |

| | |
|---|---|
| 16 | Implementation of Binary Search |
| 17 | Implementation of Menu driven Selection sort, Bubble sort, Insertion sort |
| 18 | Implementation of Menu driven Merge Sort and Quick Sort |
| A1 | Assignment on Dijkstra Algorithm to find shortest path |
| A2 | Assignment on Recursion and Storage management |

## Aim:

Implementation of push, pop, peek, display in stack.

## Explanation:

Stack is an important data structure which stores its elements in an ordered manner.
Stack is a linear, last-in first-out (LIFO) data structure in which insertion and deletion of elements are done only at one end, known as TOP of the stack.

Every stack has a variable TOP associated with it, which is used to store the address of the topmost element of the stack.

If TOP = NULL, then it indicates that the stack is empty.

If TOP = MAX-1, then the stack is full.(MAX variable is used to store maximum no. of elements a stack can store.)

Basic operations: Push, Pop, Peep.

# Algorithms:

## A) Algorithm for insertion

1. IF TOP = MAX-1 then
   PRINT "Over flow"
   Go to step 4
   ENDIF
2. SET TOP = TOP + 1
3. SET STACK [TOP] = VALUE
4. END

## B) Algorithm for deletion

1. IF TOP = NULL, then
   PRINT "UNDER FLOW"
   GO TO STEP 4
   ENDIF
2. SET VAL = STACK [TOP]
3. SET TOP = TOP - 1
4. END

**********

```
// IMPLEMENTATION OF STACK
#include <stdio.h>
#include <stdlib.h>
#define MAX 50   //Max size of stack
int stack[MAX];    //Defining stack
int top;                 //Defining top

//Function declaration
void initialize();
int isEmpty();
int isFull();
int size();
void push(int );
void pop(int*);
void peek();
void display();

int main() {
   int num, elem;
   int popped, peeked;
   initialize();
   //Enter choices
   while(1) {
      printf("\n");
      printf("* 1 -> PUSH");
      printf("\n* 2 -> POP");
      printf("\n* 3 -> PEEK");
```

```
        printf("\n* 4 -> DISPLAY");
        printf("\n* 5 -> SIZE");
        printf("\n* 6 -> EXIT\n");
        scanf("%d", &num);
        printf("\n\t");
        if(num == 1) { //PUSH
            printf("\n--Enter a number to push--   ");
            scanf("%d", &elem);
            push(elem);
        } else if(num == 2) { //POP
            pop(&popped);
        } else if(num == 3) { //PEEK
            peek(&peeked);
        } else if(num == 4) { //DISPLAY
            display();
        } else if (num == 5) { //SIZE
            printf("\n--Currently, size of stack is  =>  %d", size());
        } else if (num == 6) { //QUIT
            printf("\n\n**********************\n");
            break;
        } else { //DEFAULT
            printf("\nINVALID INPUT");
        }
    }
    return 0;
}
```

```
void initialize() {
   top = -1;
}

int isEmpty() {
   if (top == -1)
      return 1;
   return 0;
}

int size() {
   return top+1;
}

int isFull() {
   if (top == MAX-1)
      return 1;
   return 0;
}

void push(int elem) {
   if(isFull()) {
      printf("\nOVERFLOW");
      return;
   }
   top++;
   stack[top] = elem;
```

```
}

void pop(int* popped) {
   if(isEmpty()) {
      printf("\nUNDERFLOW");
      return;
   }
   *popped = stack[top];
   top--;
   printf("--Popped element is  =>  %d", *popped);
   return;
}

void peek(int* peeked) {
   if(isEmpty()) {
      printf("\nUNDERFLOW");
      return;
   }
   *peeked = stack[top];
   printf("--Top value is  =>  %d", *peeked);
   return;
}

void display() {
   int i;
   if(isEmpty()) {
      printf("\nEMPTY");
```

```
    } else {
        printf("\nElements in Stack are :  ");
        for (i=0 ; i<=top ; i++) {
            printf("%d\t", stack[i]);
        }
    }
}
```

**********

//OUTPUT

```
* 1 -> PUSH
* 2 -> POP
* 3 -> PEEK
* 4 -> DISPLAY
* 5 -> SIZE
* 6 -> EXIT
1


--Enter a number to push--   42

* 1 -> PUSH
* 2 -> POP
* 3 -> PEEK
* 4 -> DISPLAY
* 5 -> SIZE
* 6 -> EXIT
1


--Enter a number to push--   50

* 1 -> PUSH
* 2 -> POP
* 3 -> PEEK
* 4 -> DISPLAY
* 5 -> SIZE
* 6 -> EXIT
2

        --Popped element is  =>  50
* 1 -> PUSH
* 2 -> POP
* 3 -> PEEK
* 4 -> DISPLAY
* 5 -> SIZE
* 6 -> EXIT
3

        --Top value is  =>  42
```

```
* 1 -> PUSH
* 2 -> POP
* 3 -> PEEK
* 4 -> DISPLAY
* 5 -> SIZE
* 6 -> EXIT
4


Elements in Stack are :  42
* 1 -> PUSH
* 2 -> POP
* 3 -> PEEK
* 4 -> DISPLAY
* 5 -> SIZE
* 6 -> EXIT
5


--Currently, size of stack is  =>  1
* 1 -> PUSH
* 2 -> POP
* 3 -> PEEK
* 4 -> DISPLAY
* 5 -> SIZE
* 6 -> EXIT
6



************************
```

**Aim:** Evaluation of infix and prefix expression

## Theory:

Infix, Postfix and Prefix notations are three different but equivalent notations of writing algebraic expressions.

1) INFIX expression:

=> Operator is between two operand

"operand 1"  OPERATOR  "operand 2"

eg: A+B, A*(B+C)

1) POSTFIX expression:

=> Operator is after bot operand

"operand 1"  "operand 2"  OPERATOR

eg: AB+, ABC+*

1) PREFIX expression:

=> Operator is before both operand

OPERATOR  "operand 1"   "operand 2"

eg: A+B, *A+BC

**Benefits of using prefix and postfix expressions:**

- Although it is easy to write expressions using infix notation, computers find it difficult to evaluate as they need a lot of information to evaluate the expression.
- Information is needed about operator precedence, associativity rules, and brackets which overrides these rules.
- A postfix or Prefix expression does not even follow the rules of operator precedence and associativity.
- So, computers work more efficiently with expressions written using prefix and postfix notations.

**********

**Algorithms:**

Algorithm to evaluate a prefix expression

**Step 1:** Accept the Prefix expression

**Step 2:** Repeat until all characters in the prefix expression are scanned

a. Scan the prefix expression from right,one charcter at time

b. If an operand is encountered, push it on the stack

c. If an operator X is encountered, then

a. pop the top two elements from the stack as A and B

b. Evaluate A X B, where A was the topmost element and B was the element below A.

c. Push the result of evaluation on the stack

[END OF IF]

**Step 4:** SET RESULT equal to the topmost element of the stack

**Step 5:** END

---

Algorithm to evaluate infix expression:

There are two steps :-

*Step 1=> Convert infix expression to its equivalent postfix expression*

Algorithm to convert an Infix notation into postfix notation

**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 an alphabet) 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 X is encountered, then;

Repeatedly pop from stack and add each operator (popped from the stack which has the same precedence or a higher precedence than X) to the postfix expression.

If precedence of popped operator is less than that of x, push popped operator back to stack.

b. Push the operator X to the stack.

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

**Step 5:** END

*Step 1=> Evaluate postfix expression*

Algorithm to evaluate a postfix expression

**Step 1:** Add a ")" at the end of the postfix expression

**Step 2:** Scan every character of the postfix expression and repeat steps 3 and 4 until ")"is encountered

**Step 3:** IF an operand is encountered, push it on the stack

IF an operator X is encountered, then

a. pop the top two elements from the stack as A and B

b. Evaluate B X A, where A was the topmost element and B was the element below A.

c. Push the result of evaluation on the stack

[END OF IF]

**Step 4:** SET RESULT equal to the topmost element of the stack

**Step 5:** END

**Thank you**

---

# Evaluation of INFIX expression

//code

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#define MAXSTACK 100
#define SIZE 100

char stack[MAXSTACK];
int top = -1;
int topNum = -1;

void pushNum(int item) {
    if (topNum >= MAXSTACK-1) {
        printf("OVERFLOW");
        return;
    } else {
        topNum++;
        stack[topNum] = item;
    }
}

int popNum() {
    int num;
    if (topNum < 0) {
```

```
        printf("UNDERFLOW");
    } else {
        num = stack[topNum];
        topNum--;
        return num;
    }
}

void push(char item) {
    if (top >= MAXSTACK-1) {
        printf("OVERFLOW");
        return;
    } else {
        top++;
        stack[top] = item;
    }
}

char pop() {
    char item;
    if (top < 0) {
        printf("UNDERFLOW");
    } else {
        item = stack[top];
        top--;
        return item;
```

```
    }
}

int isOperator(char symbol) {
    if (symbol=='+' || symbol=='-' || symbol=='*' || symbol=='/' || symbol=='^') {
        return 1;
    } return 0;
}

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

void infixToPostfix(char infix[], char postfix[]) {
    int i=0, j=0;
    char item, x;
    strcat(infix, ")");
    push('(');
```

```
for(i=0 ; infix[i] != '\0' ; i++) {
    item = infix[i];
    if(item == '(') {
        push('(');
    } else if(isdigit(item)) {
        postfix[j++] = item;
    } else if(isOperator(item)) {
        x = pop();
        while(isOperator(x)==1 && precedence(x)>=precedence(item)) {
            postfix[j++] = x;
            x=pop();
        }
        push(x);
        push(item);
    } else if(item == ')') {
        x = pop();
        while (x != '(') {
            postfix[j++] = x;
            x = pop();
        }
    }
}
if(top > 0) {
    printf("Invalid expression");
}
postfix[j]='\0';
```

```
}

void evalPostfix(char postfix[]) {
    int i, a, b, val;
    char ch;

    for(i=0 ; postfix[i]!=')' ; i++) {
        ch = postfix[i];
        if(isdigit(ch)) {
            pushNum(ch-'0');
        } else if(ch=='+' || ch=='-' || ch=='*' || ch=='/') {
            a = popNum();
            b = popNum();
            switch(ch) {
                case '+':
                    val = b+a;
                    break;
                case '-':
                    val = b-a;
                    break;
                case '*':
                    val = b*a;
                    break;
                case '/':
                    val = b/a;
                    break;
```

```
            }
            pushNum(val);
        }
    }

    if (topNum > 0) {
        printf("Invalid Input");
    } else {
        printf("\nResult of given Infix expression is :  %d", popNum());
    }
}

int main() {
    char infix[SIZE], postfix[SIZE];
    printf("Enter infix Expression :  ");
    gets(infix);
    infixToPostfix(infix, postfix);
    puts(postfix);
    strcat(postfix, ")");
    evalPostfix(postfix);
}
//output
```

---

```
Enter infix Expression :  (2*(4-2)+7)

Result of given Infix expression is :  11


Process returned 0 (0x0)   execution time : 10.093 s
Press any key to continue.
```

---

```
Enter infix Expression :  7-4*2+3*(4+2)

Result of given Infix expression is :  17


Process returned 0 (0x0)   execution time : 18.402 s
Press any key to continue.
```

**********

# Evaluation of PREFIX expression:

//code

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAXSTACK 100
#define PREFIXSIZE 100

int stack[MAXSTACK];
int top = -1;

void push(int item) {
   if (top >= MAXSTACK-1) {
```

```
        printf("OVERFLOW");
        return;
    } else {
        top++;
        stack[top] = item;
    }
}

int pop() {
    int num;
    if (top < 0) {
        printf("UNDERFLOW");
    } else {
        num = stack[top];
        top--;
        return num;
    }
}

void evalPrefix(char prefix[]) {
    int i, a, b, val;
    char ch;

    for(i=0 ; prefix[i]!=')' ; i++) {
        ch = prefix[i];
        if(isdigit(ch)) {
            push(ch-'0');
        } else if(ch=='+' || ch=='-' || ch=='*' || ch=='/') {
            a = pop();
```

```
            b = pop();
            switch(ch) {
               case '+':
                  val = a+b;
                  break;
               case '-':
                  val = a-b;
                  break;
               case '*':
                  val = a*b;
                  break;
               case '/':
                  val = a/b;
                  break;
            }
            push(val);
         }
      }
      if (top > 0) {
         printf("Invalid input");
      } else {
         printf("\nResult of given prefix expression is :  %d\n\n", pop());
      }
   }

   int main() {
      int i;
      char prefix[PREFIXSIZE];
      printf("Enter prefix expression :  ");
```

```
    gets(prefix);
    strrev(prefix);
    strcat(prefix, ")");
    evalPrefix(prefix);
    return 0;
}
```

//output

```
Enter prefix expression :  *+23-54

Result of given prefix expression is :  5


Process returned 0 (0x0)   execution time : 6.520 s
Press any key to continue.
```

---

```
Enter prefix expression :  +-27*8/48

Result of given prefix expression is :  -5


Process returned 0 (0x0)   execution time : 11.804 s
Press any key to continue.
```

**********

**Aim:** Infix to postfix conversion using stack

## Algorithm:

Algorithm to convert an Infix notation into postfix notation

**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 an alphabet) 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 X is encountered, then;

Repeatedly pop from stack and add each operator (popped from the stack which has the same precedence or a higher precedence than X) to the postfix expression.

If precedence of popped operator is less than that of x, push popped operator back to stack.

b. Push the operator X to the stack.

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

**Step 5:** END

```
                                    ----

Enter infix Expression :  1+2*(3+5)

Corresponding postfix expression is :  1235+*+

Process returned 0 (0x0)   execution time : 8.945 s
Press any key to continue.

                                    ----

Enter infix Expression :  5-2*6+6/2

Corresponding postfix expression is :  526*-62/+

Process returned 0 (0x0)   execution time : 27.093 s
Press any key to continue.
```

# Infix to postfix:

```
//code
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#define MAXSTACK 100
#define SIZE 100

char stack[MAXSTACK];
int top = -1;

void push(char item) {
   if (top >= MAXSTACK-1) {
      printf("OVERFLOW");
      return;
   } else {
      top++;
      stack[top] = item;
   }
}
```

```
char pop() {
    char item;
    if (top < 0) {
        printf("UNDERFLOW");
    } else {
        item = stack[top];
        top--;
        return item;
    }
}

int isOperator(char symbol) {
    if (symbol=='+' || symbol=='-' || symbol=='*' || symbol=='/' || symbol=='^') {
        return 1;
    } return 0;
}

int precedence(char symbol) {
    if(symbol == '^') {
        return 3;
    } else if(symbol == '/' || symbol == '*') {
        return 2;
```

```
    } else if(symbol == '+' || symbol == '-') {
        return 1;
    } else {
        return 0;
    }
}

void infixToPostfix(char infix[], char postfix[]) {
    int i=0, j=0;
    char item, x;
    strcat(infix, ")");
    push('(');
    for(i=0 ; infix[i] != '\0' ; i++) {
        item = infix[i];
        if(item == '(') {
            push('(');
        } else if(isdigit(item)) {
            postfix[j++] = item;
        } else if(isOperator(item)) {
            x = pop();
            while(isOperator(x)==1 && precedence(x)>=precedence(item)) {
                postfix[j++] = x;
                x=pop();
```

```
            }
            push(x);
            push(item);
        } else if(item == ')') {
            x = pop();
            while (x != '(') {
                postfix[j++] = x;
                x = pop();
            }
        }
    }
    if(top > 0) {
        printf("Invalid expression");
    }
    postfix[j]='\0';
}

int main() {
    char infix[SIZE], postfix[SIZE];
    printf("Enter infix Expression :  ");
    gets(infix);
    infixToPostfix(infix, postfix);
    printf("\nCorresponding postfix expression is :  ");
```

```
    puts(postfix);
}
```

//output

```
Enter infix Expression :  1+2*(3+5)

Corresponding postfix expression is :  1235+*+

Process returned 0 (0x0)   execution time : 8.945 s
Press any key to continue.
```

---

```
Enter infix Expression :  5-2*6+6/2

Corresponding postfix expression is :  526*-62/+

Process returned 0 (0x0)   execution time : 27.093 s
Press any key to continue.
```

---

**Aim:** Implementation of Linear Queue using array

## Theory:

Queue is an important linear data structure which stores its elements in an ordered manner.

### Example:

1. People moving on an escalator. The people who got on the escalator first will be the first one to step out of it.
2. 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.
3. The elements in a queue are added at one end called the rear and removed from the other one end called the front.

### Array implementation of queue:

1. Queues can be easily represented using linear arrays.
2. Every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively.
3. Before inserting an element in the queue we must check for overflow conditions.
4. An overflow occurs when we try to insert an element into a queue that is already full, i.e. when rear = MAX – 1, where MAX specifies the maximum number of elements that the queue can hold.
5. Similarly, before deleting an element from the queue, we must check for underflow condition.
6. An underflow occurs when we try to delete an element from a queue that is already empty. If front = -1 and rear = -1, this means there is no element in the queue.

## Algorithms:

Algorithm to insert an element in a queue

**Step 1:** IF REAR=MAX-1, then;

Write OVERFLOW

Goto Step 4

[END OF IF]

**Step 2:** IF FRONT == -1 and REAR = -1, then

SET FRONT = REAR = 0

ELSE

SET REAR = REAR + 1

[END OF IF]

**Step 3:** SET QUEUE[REAR] = NUM

**Step 4:** Exit

Algorithm to delete an element from a queue

**Step 1:** IF FRONT = -1 OR FRONT > REAR, then

Write UNDERFLOW

Goto Step 2

ELSE

SET VAL = QUEUE[FRONT]

SET FRONT = FRONT + 1

[END OF IF]

**Step 2:** Exit

**Implementation of linear queue**

//code

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10

int queue[MAX];
int front = -1, rear = -1, deleted;

void insert(int );
void del(int* );
void display();

int main() {
   int i, choice, item;

   while(1) {
      printf("\n* 1. INSERT ");
      printf("\n* 2. DELETE ");
      printf("\n* 3. DISPLAY ");
      printf("\n* 4. EXIT ");
      printf("\nEnter your choice :  ");
      scanf("%d", &item);

      switch(item) {
         case 1:
```

```
            printf("\nEnter element to insert :  ");
            scanf("%d", &item);
            insert(item);
            break;
        case 2:
            del(&deleted);
            printf("\nDeleted element is :  %d", deleted);
            break;
        case 3:
            display();
            break;
        case 4:
            printf("****END****");
            exit(1);
        default:
            printf("\nInvalid input");
      }
   }
}

void insert(int item) {
   if (rear == MAX-1) {
      printf("\nQUEUE OVERFLOW");
      return;
   }
   if (front == -1) {
      front++;
```

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

void del(int* deleted) {
    if (front==-1 || front>rear) {
        printf("\nQUEUE UNDERFLOW");
        return;
    }
    *deleted = queue[front];
    front++;
}

void display() {
    int i;
    if (front==-1 || front> rear) {
        printf("\nQUEUE UNDERFLOW");
        return;
    }
    printf("Elements of queue are :  ");
    for (i=front ; i<=rear ; i++) {
        printf("%d, ", queue[i]);
    }
}
```

//output

```
* 1. INSERT
* 2. DELETE
* 3. DISPLAY
* 4. EXIT
Enter your choice :  1

Enter element to insert :  12

* 1. INSERT
* 2. DELETE
* 3. DISPLAY
* 4. EXIT
Enter your choice :  1

Enter element to insert :  35

* 1. INSERT
* 2. DELETE
* 3. DISPLAY
* 4. EXIT
Enter your choice :  1

Enter element to insert :  53

* 1. INSERT
* 2. DELETE
* 3. DISPLAY
* 4. EXIT
Enter your choice :  3
Elements of queue are :  12, 35, 53,
* 1. INSERT
* 2. DELETE
* 3. DISPLAY
* 4. EXIT
Enter your choice :  2

Deleted element is :  12
```

```
* 1. INSERT
* 2. DELETE
* 3. DISPLAY
* 4. EXIT
Enter your choice :  2

Deleted element is :  35
* 1. INSERT
* 2. DELETE
* 3. DISPLAY
* 4. EXIT
Enter your choice :  2

Deleted element is :  53
* 1. INSERT
* 2. DELETE
* 3. DISPLAY
* 4. EXIT
Enter your choice :  3

QUEUE UNDERFLOW
* 1. INSERT
* 2. DELETE
* 3. DISPLAY
* 4. EXIT
Enter your choice :  4
****END****
Process returned 1 (0x1)   execution time : 33.060 s
Press any key to continue.
```

**DSA Lab**
**Lab Experiment 5**

**Name:** Aamir Ansari **Roll no:** 01 **Batch:** A

**Aim:** Implementation of circular queue using array.

**Theory:**

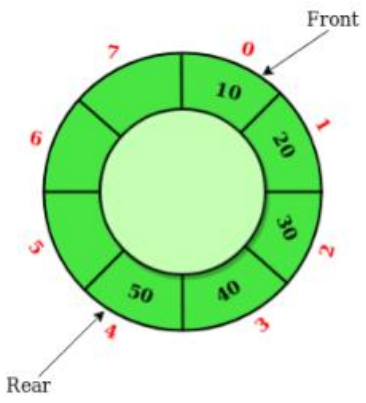
Drawbacks of linear queue:


We will explain the concept of circular queues using an example.

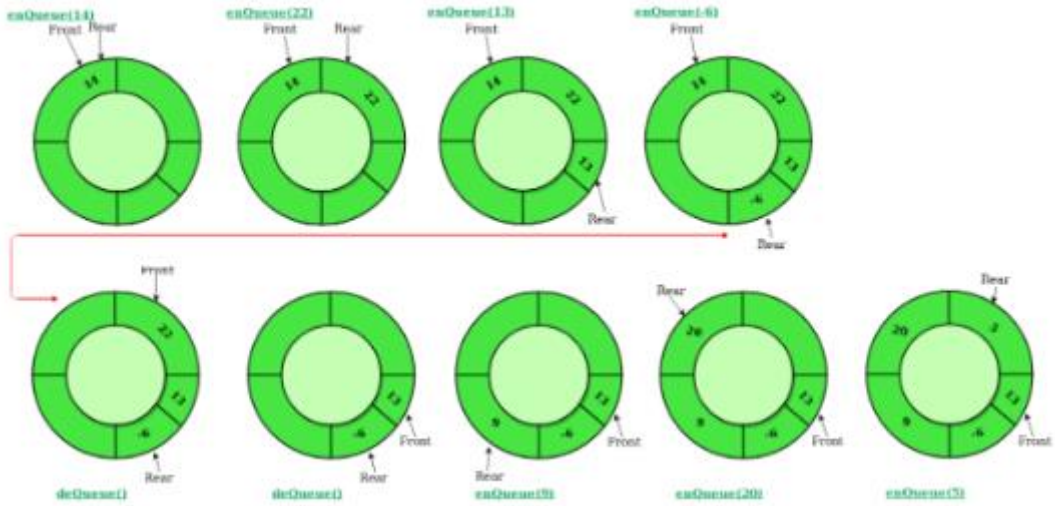
1. In this queue, front = 2 and rear = 9.
2. Now, if you want to insert a new element, it cannot be done because the space is available only at the left of the queue.
3. If rear = MAX – 1, then OVERFLOW condition exists.
4. This is the major drawback of a linear queue. Even if space is available, no insertions can be done once rear is equal to MAX – 1.
5. This leads to wastage of space. In order to overcome this problem, we use circular queues.
6. In a circular queue, the first index comes right after the last index.
7. A circular queue is full, only when front=0 and rear = Max – 1.

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 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



## Algorithms:

Algorithm to Insert an Element in a Circular Queue

**Step 1:** IF (FRONT = 0 and Rear = MAX – 1) OR (FRONT=Rear+1) Then

Write “OVERFLOW”

Goto Step 4

[END OF IF]

**Step 2:** IF FRONT = -1 and REAR = -1, then;

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

Algorithm to Delete an Element from a Circular Queue

**Step 1:** IF FRONT = -1, then 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

## Implementation of circular queue using array.

//code

```
//Implementation of circular queue
#include <stdio.h>
#include <stdlib.h>
#define MAX 5

//queue
int queue[MAX];
int rear = -1;
int front = -1;

//function prototype
void insert(int );
void del(int* deleted);
void display();

int main() {
   int choice, item, deleted;
   while(1) {
      printf("\n*1) Insert ");
      printf("\n*2) Delete ");
      printf("\n*3) Display ");
      printf("\n*4) Exit ");
      printf("\nEnter your choice :  ");
      scanf("%d", &choice);

      switch(choice) {
         case 1:
            printf("\nEnter element to insert :  ");
            scanf("%d", &item);
            insert(item);
            break;
         case 2:
            del(&deleted);
            printf("\nDeleted element is :  %d", deleted);
            break;
         case 3:
            printf("\nElements of queue are :  ");
            display();
            break;
         case 4:
            printf("****Exiting****");
            exit(1);
         default :
            printf("\nInvalid option");
      }
   }
}
```

```
void insert(int item) {
   if((front==0 && rear==MAX-1) || (front==rear+1)) {
      printf("\nOVERFLOW");
      return;
   }
   if(front==-1 && rear==-1) { //Empty queue
      front++;
      rear++;
   } else if(rear==MAX-1) { //if insertion is at last space, and 0th position is empty
      rear = 0;
   } else { //normal condition
      rear++;
   }
   queue[rear] = item;
}

void del(int* deleted) {
    if (front==-1) {
      printf("\nUNDERFLOW");
      return;
    }
    *deleted = queue[front];
    if(front == rear) { //if only remaining element is deleted
      front = -1;
      rear = -1;
    } else {
       if(front == MAX-1) { //when element is the last space is deleted and 0th position is not empty
          front = 0;
       } else { //normal condition
          front++;
       }
    }
}

void display() {
   int i;
   if(front == -1) {
      printf("UNDERFLOW");
      return;
   }
   for(i=front ; i!=rear ; i=((i+1)%MAX)) {
      printf("%d  ", queue[i]);
   }
   printf("%d", queue[rear]);
}
```

//output

```
*1) Insert
*2) Delete
*3) Display
*4) Exit
Enter your choice :  1

Enter element to insert :  5

*1) Insert
*2) Delete
*3) Display
*4) Exit
Enter your choice :  1

Enter element to insert :  10

*1) Insert
*2) Delete
*3) Display
*4) Exit
Enter your choice :  1

Enter element to insert :  15

*1) Insert
*2) Delete
*3) Display
*4) Exit
Enter your choice :  1

Enter element to insert :  20

*1) Insert
*2) Delete
*3) Display
*4) Exit
Enter your choice :  1

Enter element to insert :  25

*1) Insert
*2) Delete
*3) Display
*4) Exit
Enter your choice :  3

Elements of queue are :  5  10  15  20  25
```

```
*1) Insert
*2) Delete
*3) Display
*4) Exit
Enter your choice :  2

Deleted element is :  5
*1) Insert
*2) Delete
*3) Display
*4) Exit
Enter your choice :  2

Deleted element is :  10
*1) Insert
*2) Delete
*3) Display
*4) Exit
Enter your choice :  3

Elements of queue are :  15  20  25
*1) Insert
*2) Delete
*3) Display
*4) Exit
Enter your choice :  1

Enter element to insert :  30

*1) Insert
*2) Delete
*3) Display
*4) Exit
Enter your choice :  3

Elements of queue are :  15  20  25  30
*1) Insert
*2) Delete
*3) Display
*4) Exit
Enter your choice :  4
****Exiting****
Process returned 1 (0x1)   execution time : 38.701 s
Press any key to continue.
```

**DSA LAB**
**Experiment number 06**

**Name:** Aamir Ansari **Batch:** A **Roll no.** 01

**Aim:** Implementation of singly linked list

## Theory:

A singly linked list is a type of linked list that is unidirectional, that is, it can be traversed in only one direction from head to the last node (tail).
Each element in a linked list is called a **node**. A single node contains data and a pointer to the next node which helps in maintaining the structure of the list.

## Algorithms:

**1.CREATE LIST**

Step 1:[INITIALIZE] n /*number of nodes to be entered*?
Step 2: [INITIALIZE] node new_node[n]
Step 3:SET START2=new_node[0]
Step 4: Repeat step 5&6 while i<n
Step 5:SET newnode[i]->data
Step 6:SET newnode[i]->next = newnode[i+1]
Step 7: EXIT

**2.INSERT**

a)At a position:

Step 1: [INITIALIZE] new , p
Step 2:SET new->data=data;
Step 3: IF position==1
Step 4: SET new->next=START
Step 5: SET START=new[END OF IF]
Step 6:SET p=START
Step 7:Repeat step 8 & 9 while i<position -1
Step 8: SET p=p->next
Step 9: SET i++
Step 10:IF p==NULL
PRINT "There are less elements"
Step 11:ELSE
SET new->next=p->next
SET p->next=new
Step 12:EXIT

b)After a given Value:

Step 1: [INITIALIZE] New_Node ,ptr ,preptr
Step 2: SET New_Node->data= data
Step 3: SET ptr=START

Step 4: SET preptr=ptr
Step 5: Repeat step 6&7 while preptr->data != val
Step 6: SET preptr=ptr
Step 7: SET ptr=ptr->next
Step 8: SET preptr->next=New_Node
Step 9: SET New_Node->next=ptr
Step 10:EXIT

c)Before a given value:
Step 1: [INITIALIZE] New_Node ,ptr ,preptr
Step 2: SET New_Node->data= data
Step 3: SET ptr=START
Step 4: SET preptr=ptr
Step 5: Repeat step 6&7 while ptr->data != val
Step 6: SET preptr=ptr
Step 7: SET ptr=ptr->next
Step 8: SET preptr->next=New_Node
Step 9: SET New_Node->next=ptr
Step 10:EXIT

d)At the beginning
Step 1: [INITIALIZE] New_node
Step 2: IF START == NULL
SET START = New_node
SET START->next = NULL
[END OF IF]
Goto Step 6
Step 3: SET New_Node->DATA = VAL
Step 4: SET New_Node->next = START
Step 5: SET START = New_Node
Step 6: EXIT

e)At the end
Step 1: [INITIALIZE] New_node,last
Step 2:SET New_Node->data=val
Step 3: IF START == NULL
SET START = New_node
SET START->next = NULL
[END OF IF]
Goto Step 6
Step 4: SET last = START;
Step 5:Repeat Step 6 while last->next != NULL
Step 6: SET last = last->next; [END OF LOOP]
Step 7:SET last->next = New_Node;
Step 8:SET New_Node->next = NULL;
Step 9: EXIT

**3.DELETION**
a)Value at a particular Position
Step 1: [INITIALIZE] ptr , preptr
Step 2: IF START == NULL

PRINT "Linked list is already empty"
Goto Step 13
Step 3:IF START->next == NULL
free(START)
START = NULL
Goto step 10
Step 4: SET count = 1
Step 5:SET ptr= START
Step 6: SET preptr = ptr
Step 7:Repeat step 8 to 10 while count < position
Step 8: SET preptr = ptr
Step 9: SET ptr = ptr->next
Step 10: SET count++
Step 11: IF count == 1
START = ptr->next
ptr->next = NULL
free(ptr)
Step 12:ELSE
preptr->next = ptr->next
ptr->next = NULL
free(ptr)
Step 13:EXIT

b)Value at the beginning
Step 1: [INITIALIZE] New_node
Step 2: IF START == NULL
PRINT "Linked list is already empty"
Goto Step 6
Step 3: SET New_Node=START
Step 4: SET START = START->next
Step 5: free(New_Node)
Step 6: EXIT

c)At the end
Step 1: [INITIALIZE] ptr , preptr
Step 2: IF START == NULL
PRINT "Linked list is already empty"
Goto Step 6
Step 3:IF START->next == NULL
free(START)
START = NULL
Goto step 10
Step 4:SET ptr= START
Step 5: Repeat step 6&7 while ptr->next != NULL
Step 6: preptr = ptr
Step 7: ptr = ptr->next
Step 8: SET preptr->next = NULL
Step 9: free(ptr)
Step 10:EXIT

d)After a particular value
Step 1: [INITIALIZE] temp ,ptr ,preptr

Step 2: SET ptr=START
Step 3: SET preptr=ptr
Step 4: Repeat step 5&6 while preptr->data != val
Step 5: SET preptr=ptr
Step 6: SET ptr=ptr->next
Step 7: SET temp=ptr
Step 8: SET preptr->next=temp->next
Step 9: free(temp)
Step 10:EXIT

e)Before a particular value
Step 1: [INITIALIZE] new_Node ,ptr ,preptr
Step 2: SET ptr=START
Step 3: SET preptr=ptr
Step 4: Repeat step 5&6 while ptr->data != val
Step 5: SET preptr=ptr
Step 6: SET ptr=ptr->next
Step 7: SET preptr->next=new_Node
Step 8:SET new_Node->next=ptr
Step 9:EXIT

**4.UPDATE**
a)Value at a given Position
Step 1: IF START == NULL
PRINT "Linked list is already empty"
Goto Step 7
Step 2: SET count = 1
Step 3: Repeat step 4&5 while count != position
Step 4: ptr = ptr->next
Step 5: count++
Step 6:SET ptr->data = data
Step 7: EXIT

b)Value at the beginning
Step 1: IF START == NULL
PRINT "Linked list is already empty"
Goto Step 3
Step 2: SET START->data = data
Step 3:EXIT

c)At the end
Step 1: IF START == NULL
PRINT "Linked list is already empty"
Goto Step 3
Step 2: [INITIALIZE] node ptr=START
Step 3:Repeat step 4 while ptr->next != NULL
Step 4: ptr = ptr->next
Step 5:SET ptr->data = data
Step 6:EXIT

d)After a particular value
Step 1: IF START == NULL

PRINT "Linked list is already empty"
Goto Step 3
Step 2: [INITIALIZE] node ptr=START , postptr
Step 3:Repeat step 4 while ptr->next != val
Step 4: ptr = ptr->next
Step 5: IF ptr->next == NULL
PRINT "There is no element after this!"
Step 6:ELSE
SET postptr = ptr->next
SET postptr->data = data
Step 7: EXIT

e)Before a particular value
Step 1: IF START == NULL
PRINT "Linked list is already empty"
Goto Step 3
Step 2: [INITIALIZE] node ptr=START , preptr
Step 3:Repeat step 4 while ptr->next != val
Step 4: ptr = ptr->next
Step 5: IF ptr->next == NULL
PRINT "There is no element after this!"
Step 6:ELSE
SET preptr = ptr
SET ptr = ptr->next
SET count++
Step 7: SET preptr->data = data;
Step 8:EXIT

**5.SEARCH**
Step 1: SET PTR = START
Step 2: Repeat Step 3 while PTR != NULL
Step 3: IF VAL = PTR->DATA
PRINT 'ELEMENT FOUND'
Go To Step 5
ELSE
SET PTR = PTR->NEXT
[END OF IF]
Step 4: PRINT 'ELEMENT NOT FOUND'
Step 5: EXIT

**6.REVERSE**
Step 1:[INITIALIZE] prev, ptr, next
Step 2:SET prev=NULL
Step 3:SET ptr=START
Step 4: Repeat step 5 to 8 while ptr!=NULL
Step 5: SET next=ptr->next
Step 6: SET ptr->next=prev
Step 7: SET prev=ptr
Step 8: SET ptr=next
Step 9:SET START=prev

Step 10:EXIT

**7.COUNT NODES**

Step 1: INITIALIZE count = 0,node current = START
Step 2:Repeat step3&4 while current != NULL
Step 3: SET count++
Step 4: SET current = current->next
Step 5 :RETURN count
Step 6:EXIT

**8.DISPLAY**

Step 1:Repeat step 2&3 while START != NULL
Step 2: PRINT (START->data)
Step 3: SET START = START->next
Step 4: EXIT

**9.MERGE**

Step 1: [INITIALIZE ] ptr , ptr2
Step 2: SET ptr = START
Step 3:Repeat step 4 while ptr->next != NULL
Step 4: SET ptr = ptr->next
Step 5: SET ptr->next = START2
Step 6:[INITIALIZE] node traverse , min , temp
Step 7:Repeat step 8&9 while START->next
Step 8: SET min = START
Step 9: SET traverse = START->next
Step 10:Repeat step 11&12 while traverse is true
Step 11: IF min->data > traverse->data
SET min = traverse
Step 12: SET traverse = traverse->next
Step 13: SET temp = START->data
Step 14: SET START->data = min->data
Step 15: SET min->data = temp
Step 16: SET START = START->next
Step 17:EXIT

**10.SORT**

Step 1:[INITIALIZE] node traverse , min , temp
Step 2:Repeat step 3&4 while START->next
Step 3: SET min = START
Step 4: SET traverse = START->next
Step 5:Repeat step 6&7 while traverse is true
Step 6: IF min->data > traverse->data
SET min = traverse
Step 7: SET traverse = traverse->next
Step 8: SET temp = START->data
Step 9: SET START->data = min->data
Step 10: SET min->data = temp
Step 11: SET START = START->next
Step 12:EXIT

**11.CONCATENATE TWO SLLS.**

Step 1: [INITIALIZE ] ptr , ptr2
Step 2: SET ptr = START
Step 3:Repeat step 4 while ptr->next != NULL
Step 4:  SET ptr = ptr->next
Step 5: SET ptr->next = START2
Step 6:EXIT

## Implementation of Singly linked list:

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


struct node {   // declaration for main linked list
   int data;
   struct node *next;
};
//Start node
struct node *start = NULL;


struct nodeTwo {   // Declaration for secondary linked list
   int dataTwo;
   struct nodeTwo *nextTwo;
};
// Start node of secondary linked list
struct nodeTwo *startTwo = NULL;

void secondLinkedList() {     // Initialises second linked list with static values
   // declare nodes
   struct nodeTwo *newNodeOne;
   struct nodeTwo *newNodeTwo;
   struct nodeTwo *newNodeThree;
   // allocates memory for nodes
   newNodeOne = (struct nodeTwo *)malloc(sizeof(struct nodeTwo));
   newNodeTwo = (struct nodeTwo *)malloc(sizeof(struct nodeTwo));
   newNodeThree = (struct nodeTwo *)malloc(sizeof(struct nodeTwo));
   // enter data and link the nodes
   startTwo = newNodeOne;
   newNodeOne->dataTwo = 4;
   newNodeOne->nextTwo = newNodeTwo;

   newNodeTwo->dataTwo = 8;
   newNodeTwo->nextTwo = newNodeThree;

   newNodeThree->dataTwo = 12;
```

```
    newNodeThree->nextTwo = NULL;

}

void insertAtBegining(int val) {    // Inserts node at the begining
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = val;
    if (start == NULL) {    // when 0 nodes are present
        start = newNode;
        start->data = newNode->data;
        start->next = NULL;
        return;
    }
    newNode->next = start;
    start = newNode;
}

void insertAtEnd(int val) {     // Inserts at the end
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = val;
    if (start == NULL) {    // Entering first node
        start = newNode;
        start->data = newNode->data;
        start->next = NULL;
        return;
    } else {
        struct node *ptr;
        ptr = start;

        while (ptr->next != NULL) {
            ptr = ptr->next;
        }
        ptr->next = newNode;
        newNode->next = NULL;
    }

}
```

```
void insertAfterNum(int toInsert, int val) {    // Inserts after a value
   struct node *newNode;
   struct node *temp;  // to store address of next pointer
   struct node *ptr;   // traversing pointer
   newNode = (struct node *)malloc(sizeof(struct node));
   newNode->data = toInsert;
   ptr = start;
   while (ptr->data!=val) {    //traverse upto val
      ptr = ptr->next;
   }

   temp = ptr->next;       // store address of next node
   ptr->next = newNode;    // change address to address of new node
   newNode->next = temp;   // set address of new node to the following node
   return;
   printf("\nValue is not present!");
}

void insertBeforeNum(int toInsert, int val) {   // Insert before a value
   if (start == NULL) {
      printf("\nLinked list is empty!");
      return;
   }

   struct node *newNode;
   struct node *ptr;
   struct node *prePtr;
   ptr = start;

   newNode = (struct node *)malloc(sizeof(struct node));
   newNode->data = toInsert;

   if  (start->data == val) {  // Inserting before first node
      start = newNode;
      newNode->next = ptr;
      return;
   }

   while(ptr->data != val) {   // Traversing
      prePtr = ptr;
```

```
        ptr = ptr->next;
    }
    // Inserting before any node
    prePtr->next = newNode;
    newNode->next = ptr;
}

void insertAfterPos(int toInsert, int pos) {    // Insert after a given position
    struct node *newNode;
    struct node *temp;  // to store address of next pointer
    struct node *ptr;   // traversing pointer
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = toInsert;
    ptr = start;
    int count = 1;
    while (count != pos) {    // traverse upto pos
        ptr = ptr->next;
        count++;
    }
    temp = ptr->next;       // store address of next node
    ptr->next = newNode;    // change address to address of new node
    newNode->next = temp;   // set address of new node to the following node
    return;
}

void deleteAtBegining() {   // Delete element at the begining
    struct node *ptr;
    ptr = start;
    if (start == NULL) {
        printf("\nLinked list is empty!");
        return;
    }
    if (start->next == NULL) {  // Deleting only remaining first node
        printf("\nDeleted element is :  %d", ptr->data);
        start = NULL;
        return;
    }

    // Deleting any node
    printf("\nDeleted element is :  %d", ptr->data);
```

```
    ptr = ptr->next;
    start->data = ptr->data;
    start->next = ptr->next;
}

void deleteAtEnd() {    // Deletes element at the end
    if (start == NULL) {
        printf("\nLinked list is empty!");
    }
    struct node *ptr;
    struct node *prePtr;
    ptr = start;
    if (start->next == NULL) {  // deleting only remaining node
        printf("\nDeleted element is :  %d", ptr->data);
        start = NULL;
        return;
    }
    while (ptr->next != NULL) { // Traversing
        prePtr = ptr;
        ptr = ptr->next;
    }
    printf("\nDeleted element is :  %d", ptr->data);
    prePtr->next = NULL;
}

void deleteAtPos(int pos) {     // Deltes node after entered position
    if (start == NULL) {
        printf("\nLinked list is empty!");
        return;
    }
    struct node *ptr;
    struct node *prePtr;
    int count = 1;
    ptr = start;
    prePtr = ptr;
    if (start->next == NULL) {  // deleting only remaining node
        printf("\nDeleted element is :  %d", ptr->data);
        start = NULL;
        return;
    }
```

```
    while (count < pos) {   // Traversing
        prePtr = ptr;
        ptr = ptr->next;
        count++;
    }
    if (count == 1) {   // Deleting first node
        printf("\nDeleted Element is :  %d", ptr->data);
        start = ptr->next;
        ptr->next = NULL;
        free(ptr);
    } else {    // Deleting any other node
        printf("\nDeleted Element is :  %d", ptr->data);
        prePtr->next = ptr->next;
        ptr->next = NULL;
        free(ptr);
    }
}

void deleteAfterVal(int val) {   // Deletes after a given value
    if (start == NULL) {
        printf("\nLinked list is empty!");
        return;
    }
    struct node *ptr = start;
    struct node *postPtr;

    while (ptr->data != val) {  // Traversing
        ptr = ptr->next;
    }
    if (ptr->next == NULL) {
        printf("\nThere is no element after this!");
    } else {
        printf("\nDeleted element is :  %d", ptr->next->data);
        postPtr = ptr->next;
        ptr->next = postPtr->next;
        postPtr->next = NULL;
    }
}

void deleteBeforeVal(int val) {     // Deletes a node before a given value
```

```
    if (start == NULL) {
        printf("\nLinked list is empty!");
        return;
    }
    struct node *ptr = start;
    struct node *prePtr = ptr;

    if (start->data == val) {
        printf("\nNo node before this!");
        return;
    }

    if (start->next->data == val) {     // If first node is to be deleted
        printf("\nDeleted element is :  %d", start->data);
        start = start->next;
        return;
    }

    ptr = start;
    prePtr = ptr;
    while (ptr->next->data != val) {
        prePtr = ptr;
        ptr = ptr->next;
    }

    // Deleting any other node
    printf("\nDeleted element is :  %d", ptr->data);
    prePtr->next = ptr->next;
    ptr->next = NULL;
    free(ptr);

}

void updateAtBeginning (int val) {      // Updates value at the start
    if (start == NULL) {
        printf("\nLinked list is empty!");
        return;
    }
    start->data = val;
}
```

```
void updateAtEnd (int val) {    // Updates value at the end
   if (start == NULL) {
      printf("\nLinked list is empty!");
      return;
   }
   struct node *ptr = start;
   while (ptr->next != NULL) {
      ptr = ptr->next;
   }
   ptr->data = val;
}

void updateAtPos(int toInsert, int pos) {    // Updates value at the given position
   if (start == NULL) {
      printf("\nLinked list is empty!");
      return;
   }
   int count = 1;
   struct node *ptr = start;
   while (count != pos) {
      ptr = ptr->next;
      count++;
   }
   ptr->data = toInsert;
}

void updateAfterVal(int toInsert, int val) {    // Updates after entered value is encountered
   if (start == NULL) {
      printf("\nLinked list is empty!");
      return;
   }
   struct node *ptr = start;
   struct node *postPtr;

   while (ptr->data != val) {    // Traversing
      ptr = ptr->next;
   }
   if (ptr->next == NULL) {     // If the value is of last node
      printf("\nThere is no element after this!");
```

```
    } else {    // Update any other node
        postPtr = ptr->next;
        postPtr->data = toInsert;
    }
}

void updateBeforeVal(int toInsert, int val) {    // Updates before entered value is encounterd
    if (start == NULL) {
        printf("\nLinked list is empty!");
        return;
    }
    struct node *ptr = start;
    struct node *prePtr;
    int count = 0;
    while (ptr->data != val) {    // Traverse
        prePtr = ptr;
        ptr = ptr->next;
        count++;
    }
    if (count == 0) {    // If value is of first node
        printf("\nThere is no element before this!");
        return;
    }
    // Update any other node
    prePtr->data = toInsert;
}

void search(int val) {  // Search for element in the array
    struct node *ptr;
    int count = 0;
    ptr = start;
    if (ptr == NULL) {
        printf("\nList is empty");
        return;
    }
    while (ptr->next != NULL) {
        if (val == ptr->data) {
            printf("\n%d  is present on node index  :  %d", val, count);
            return;
        }
```

```
        ptr = ptr->next;
        count++;
    }
    printf("\nElement not found!");
}

void reverse() {    // Reverses the list
    struct node *previousNode, *currentNode, *nextNode;
    previousNode = NULL;
    currentNode = nextNode = start;
    while (nextNode != NULL) {
        nextNode = nextNode->next;
        currentNode->next = previousNode;
        previousNode = currentNode;
        currentNode = nextNode;
    }
    start = previousNode;
}

void countNodes() {    // Count nodes in the list
    struct node *ptr = start;
    int count = 1;
    while (ptr->next != NULL) {
        ptr = ptr->next;
        count++;
    }
    printf("There are  %d  nodes", count);
}

void display() {  // traverse through the list
    struct node* ptr;
    ptr = start;
    if (ptr == NULL) {
        printf("\nList is empty!");
        return;
    }
    printf("\n");
    while (ptr->next != NULL) {
        printf("%d  ", ptr->data);
        ptr = ptr->next;
```

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

void sort() {    // Sorts the list
    struct node *i = start;
    struct node *j = NULL;
    int temp;
    for (i = start ; i != NULL ;  i=i->next) {
        for (j = i->next ; j != NULL ; j = j->next) {
            if (i->data > j->data) {
                temp = i->data;
                i->data = j->data;
                j->data = temp;
            }
        }
    }
}

void concat() {
    struct node *ptr;
    struct nodeTwo *ptrTwo;
    ptr = start;
    while (ptr->next != NULL) {
        ptr = ptr->next;
    }
    ptr->next = (struct node *)startTwo;
}

void displayListTwo() {

    struct nodeTwo* ptr;
    ptr = startTwo;
    if (ptr == NULL) {
        printf("\nList is empty!");
        return;
    }
    printf("\n");
    while (ptr->nextTwo != NULL) {
        printf("%d  ", ptr->dataTwo);
```

```
        ptr = ptr->nextTwo;
    }
    printf("%d ", ptr->dataTwo);
}




int main() {
    int choice, item, pos, val;
    // displayListTwo();

    while (1) {
        printf("\n*1  Insert at the beginning");
        printf("\n*2  Insert at the end");
        printf("\n*3  Insert after position");
        printf("\n*4  Insert after a given value");
        printf("\n*5  Insert before given value");
        printf("\n*6  Delete at a particular position");
        printf("\n*7  Delete at beginning");
        printf("\n*8  Delete value at end");
        printf("\n*9  Delete after a particular value");
        printf("\n*10 Delete before a particular value");
        printf("\n*11 Update the value of given position");
        printf("\n*12 Update value at the beginning");
        printf("\n*13 Update value at the end");
        printf("\n*14 Update after a particular value");
        printf("\n*15 Update before a particular value");
        printf("\n*16 Search");
        printf("\n*17 Reverse");
        printf("\n*18 Count Nodes");
        printf("\n*19 Display");
        printf("\n*20 Sort");
        printf("\n*21 Concat");
        printf("\n*22 Merge");
        printf("\n*23 EXIT");
        printf("\n");
        printf("\nEnter your choice :  ");
        scanf("%d", &choice);
```

```
switch(choice) {

    case 1:
        printf("\nEnter an element to add :  ");
        scanf("%d", &item);
        insertAtBegining(item);
        break;

    case 2:
        printf("\nEnter an element to add :  ");
        scanf("%d", &item);
        insertAtEnd(item);
        break;

    case 3:
        printf("\nEnter an element to add :  ");
        scanf("%d", &item);
        printf("\nEnter position after which to add :  ");
        scanf("%d", &pos);
        insertAfterPos(item, pos);
        break;

    case 4:
        printf("\nEnter an element to add :  ");
        scanf("%d", &item);
        printf("\nEnter value after which to add :  ");
        scanf("%d", &val);
        insertAfterNum(item, val);
        break;

    case 5:
        printf("\nEnter an element to add :  ");
        scanf("%d", &item);
        printf("\nEnter value before which to add :  ");
        scanf("%d", &val);
        insertBeforeNum(item, val);
        break;

    case 6:
        printf("\nEnter position from where to delete :  ");
```

```
    scanf("%d", &item);
    deleteAtPos(item);
    break;

case 7:
    deleteAtBegining();
    break;

case 8:
    deleteAtEnd();
    break;

case 9:
    printf("\nEnter value after which to delete :  ");
    scanf("%d", &item);
    deleteAfterVal(item);
    break;

case 10:
    printf("\nEnter value before which to delete :  ");
    scanf("%d", &item);
    deleteBeforeVal(item);
    break;

case 11:
    printf("\nEnter an element to update :  ");
    scanf("%d", &item);
    printf("\nEnter value at which to update :  ");
    scanf("%d", &pos);
    updateBeforeVal(item, pos);
    break;

case 12:
    printf("\nEnter an element to update :  ");
    scanf("%d", &item);
    updateAtBeginning(item);
    break;

case 13:
    printf("\nEnter an element to update :  ");
```

```
    scanf("%d", &item);
    updateAtEnd(item);
    break;

case 14:
    printf("\nEnter an element to update :  ");
    scanf("%d", &item);
    printf("\nEnter value after which to update :  ");
    scanf("%d", &val);
    updateAfterVal(item, val);
    break;

case 15:
    printf("\nEnter an element to update :  ");
    scanf("%d", &item);
    printf("\nEnter value before which to update :  ");
    scanf("%d", &val);
    updateBeforeVal(item, val);
    break;

case 16:
    printf("\nEnter elment to search ");
    scanf("%d", &item);
    search(item);
    break;

case 17:
    reverse();
    break;

case 18:
    countNodes();
    break;

case 19:
    printf("\nEnlements in the list are :");
    display();
    break;

case 20:
```

```
            sort();
            break;

        case 21:
            printf("List 1 :  ");
            display();
            printf("\nList 2 :  ");
            secondLinkedList();
            displayListTwo();
            concat(item);
            printf("\nList after concatenation :  ");
            display();
            break;

        case 22:
            printf("List 1 :  ");
            display();
            printf("\nList 2 :  ");
            secondLinkedList();
            displayListTwo();
            concat();
            sort();
            printf("\nList after merging :  ");
            display();
            break;

        case 23:
            printf("\n***EXITING***\n");
            exit(1);
            break;
        default:
            printf("INVALID INPUT");
      }
   }

   return 0;
}
```

//output

```
*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT

Enter your choice :  1

Enter an element to add :  5

*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
```

```
Enter your choice :  2

Enter an element to add :  10

*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT

Enter your choice :  2

Enter an element to add :  15

*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
```

```
Enter your choice :  2

Enter an element to add :  15

*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT

Enter your choice :  19

Enlements in the list are :

5  10  15
*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
```

```
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT

Enter your choice :  3

Enter an element to add :  20

Enter position after which to add :  3

*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT

Enter your choice :  4

Enter an element to add :  25

Enter value after which to add :  20

*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
```

```
Enter your choice :  4

Enter an element to add :  25

Enter value after which to add :  20

*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT

Enter your choice :  19

Enlements in the list are :

5  10  15  20  25
```

```
*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT

Enter your choice :  6

Enter position from where to delete :  2
```

```
*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT

Enter your choice :  7

*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT
```

```
*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT

Enter your choice :  19

Enlements in the list are :

15  20
```

```
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT

Enter your choice :  9

Enter value after which to delete :  15

*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT
```

```
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT

Enter your choice :  12

Enter an element to update :  20

*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT
```

```
Enlements in the list are :

20
*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT

Enter your choice :  2

Enter an element to add :  15
```

```
*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT

Enter your choice :  2

Enter an element to add :  10

*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
```

```
Enter your choice :  19

Enlements in the list are :

20  15  10
*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT

Enter your choice :  12

Enter an element to update :  40
```

```
*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT

Enter your choice :  19

Enlements in the list are :

40  15  10
```

```
*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT

Enter your choice :  16

Enter elment to search 15
```

```
*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT

Enter your choice :  18
There are  3  nodes
*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT
```

```
*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT

Enter your choice :  19

Enlements in the list are :

10  15  40
```

```
*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT

Enter your choice :  17

*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT
```

```
40  15  10
*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT

Enter your choice :  21

Enter an element to Concat :  55
```

```
*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT

Enter your choice :  19

Enlements in the list are :

55  40  15  10
```

```
Enlements in the list are :

55  40  15  10
*1  Insert at the beginning
*2  Insert at the end
*3  Insert after position
*4  Insert after a given value
*5  Insert before given value
*6  Delete at a particular position
*7  Delete at beginning
*8  Delete value at end
*9  Delete after a particular value
*10 Delete before a particular value
*11 Update the value of given position
*12 Update value at the beginning
*13 Update value at the end
*14 Update after a particular value
*15 Update before a particular value
*16 Search
*17 Reverse
*18 Count Nodes
*19 Display
*20 Sort
*21 Concat
*22 EXIT

Enter your choice :  22

***EXITING***

Process returned 1 (0x1)   execution time : 707.504 s
Press any key to continue.
```

**DSA Lab**
**Experiment number 07**

**Name:** Aamir Ansari **Batch:** A **Roll no:** 01

**Aim:** Implementation of Stack using Singly linked list

## Algorithms:

Algorithm to Push an element onto stack:

Step 1: [INITIALIZE] New_node
Step 2: SET New_Node->Data = START
Step 3: New_Node->Next = TOP
Step 4: SET TOP = New_Node
Step 5: EXIT

Algorithm to Push an element onto stack:

Step 1: IF START == NULL
PRINT "Stack is already empty"
Goto Step 7
Step 2: [INITIALIZE] ptr
Step 3: SET ptr = TOP
Step 4: SET TOP = ptr->Next
Step 5: free(ptr)
Step 6: EXIT

Algorithm to Display element from stack:

Step 1: IF START == NULL
PRINT "Stack is already empty"
Goto Step 8
Step 2: [INITIALIZE] ptr
Step 3: SET ptr = TOP
Step 4: Repeat steps 5, 6 WHILE ptr->Next != NULL
Step 5: PRINT ptr->Data
Step 6: SET ptr = ptr-> Next
[End of loop]
Step 7: PRINT ptr -> Data
Step 8: EXIT

Algorithm to Peek element from stack:

Step 1: IF START == NULL
PRINT "Stack is already empty"
Goto Step 3
Step 2: PRINT TOP -> Data
Step 3: EXIT

Algorithm to find Size of stack:

Step 1: IF START == NULL
PRINT 0
[End If]
Step 2: [INITIALIZE] ptr, count=1
Step 3: SET ptr = TOP
Step 4: Repeat step 5 WHILE ptr->Next != NULL
Step 5: SET count = count + 1
[End of loop]
Step 7: PRINT count
Step 8: EXIT

**Implementation of stack using linked list:**

//code

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

// Implementation of stack using linked list;

// declaration of linked list
struct node {
    int data;
    struct node *next;
};
// declaration of top;
struct node *TOP = NULL;


void push(int val) { // push value on to stack
    // declaring node and allocating memory
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));

    // inserting value in the node
    newNode->data = val;

    // Linking nodes of the list
    newNode->next = TOP;

    // shifting TOP
    TOP = newNode;
}

void pop() { // pop element from the stack
    // declaring traversing ptr
    struct node *ptr;
    ptr = TOP;

    if (TOP == NULL) { // Check if the stack is empty
        printf("\nStack is empty");
        return;
    }

    // printing the top element
    printf("\nPopped element is :  %d", TOP->data);
```

```
    // shifting top to second element
    TOP = ptr->next;

    // deleting node from the memory
    free(ptr);
}

void peek() { // prints top element from the stack

    if (TOP == NULL) { // checks if stack is empty
        printf("\nStack is empty!");
        return;
    }
    // prints the top most element
    printf("\nTop element of stack is :  %d", TOP->data);
}

int size() {
    // declaring a traversing pointer
    struct node *ptr;
    ptr = TOP;
    int count = 1;

    if (TOP == NULL) { // if the stack is empty
        return 0;
    }

    while (ptr->next != NULL) { // traverse the stack and increase the counter
        ptr = ptr->next;
        count++;
    }
    return count;
}

void display() { // display the complete stack
    // declaring traversing pointer
    struct node *ptr;
    ptr = TOP;

    if (TOP == NULL) { // check if the list is empty
        printf("\nStack is empty!");
        return;
    }
```

```
    printf("\nElements in the stack are :  ");
    while (ptr->next != NULL) { // traverse while printing
        printf("%d  ", ptr->data);
        ptr = ptr->next;
    }
    printf("%d", ptr->data);
}

int main() {
    int choice, val;

    while (1) {
        printf("\n*1. PUSH");
        printf("\n*2. POP");
        printf("\n*3. PEEK");
        printf("\n*4. SIZE");
        printf("\n*5. DISPLAY");
        printf("\n*6. EXIT");
        printf("\nEnter your choice :  ");
        scanf("%d", &choice);

        switch (choice) {

        case 1:
            printf("\nEnter an element to push :  ");
            scanf("%d", &val);
            push(val);
            break;
        case 2:
            pop();
            break;
        case 3:
            peek();
            break;
        case 4:
            printf("\nSize of stack is :  %d", size());
            break;
        case 5:
            display();
            break;
        case 6:
            printf("\n *** E X I T I N G ***");
            exit(1);
        default:
            printf("\nINVALID INPUT");
```

```
        }
    }
    return 0;
}
```

// output

```
*1. PUSH
*2. POP
*3. PEEK
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  1

Enter an element to push :  5

*1. PUSH
*2. POP
*3. PEEK
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  1

Enter an element to push :  10

*1. PUSH
*2. POP
*3. PEEK
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  1

Enter an element to push :  15

*1. PUSH
*2. POP
*3. PEEK
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  5

Elements in the stack are :  15  10  5
*1. PUSH
*2. POP
*3. PEEK
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  4

Size of stack is :  3
```

```
*1. PUSH
*2. POP
*3. PEEK
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  2

Popped element is :  15
*1. PUSH
*2. POP
*3. PEEK
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  3

Top element of stack is :  10
*1. PUSH
*2. POP
*3. PEEK
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  5

Elements in the stack are :  10  5
*1. PUSH
*2. POP
*3. PEEK
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  2

Popped element is :  10
*1. PUSH
*2. POP
*3. PEEK
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  2

Popped element is :  5
```

```
*1. PUSH
*2. POP
*3. PEEK
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  2

Popped element is :  5
*1. PUSH
*2. POP
*3. PEEK
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  5

Stack is empty!
*1. PUSH
*2. POP
*3. PEEK
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  1

Enter an element to push :  100

*1. PUSH
*2. POP
*3. PEEK
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  5

Elements in the stack are :  100
*1. PUSH
*2. POP
*3. PEEK
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  6

 *** E X I T I N G ***
```

**DSA Lab**
**Experiment number 07**

**Name:** Aamir Ansari **Batch:** A **Roll no:** 01

**Aim:** Implementation of queue using Singly linked list

## Algorithms:

**Algorithm to insert in the queue**

Step 1: [INITIALIZE] New_node
Step 2: SET New_Node->Data = VAL
Step 3: IF Front = NULL AND Rear = NULL THEN
SET Front = New_Node
SET Rear = New_Node
SET New_Node->Next = NULL
[END IF]
Step 4: ELSE
Rear->Next = New_Node
Rear = New_Node
New_Node -> Next = NULL
[END ELSE]
Step 5: EXIT

**Algorithm to delete from queue**

Step 1: [INITIALIZE] ptr
Step 2: IF Front = NULL AND Rear = NULL THEN
PRINT "Queue is already empty"
Goto Step 6
[END IF]
Step 3: Front = ptr -> Next
Step 4: Free(ptr)
Step 5: IF Front = NULL
SET Rear = NULL
[END IF]
Step 6: EXIT

**Algorithm to Display front of the queue**

Step 1: IF Front = NULL AND Rear = NULL THEN
PRINT "Queue is already empty"
Goto Step 6
[END IF]
Step 2: PRINT Front -> Data

**Algorithm to display size of queue**

Step 1: IF Front = NULL AND Rear = NULL THEN
PRINT 0
GOTO Step 9
[END IF]
Step 2: SET Count = 1
Step 3: [INITIALIZE] ptr
Step 4: ptr = Front
Step 5: REPEAT Steps 6, 7 WHILE ptr -> Next != NULL
Step 6: ptr = ptr -> Next
Step 7: Count = Count + 1
[END LOOP]
Step 8: PRINT Count
Step 9: EXIT

**Algorithm to Display elements of the queue**

Step 1: IF Front = NULL AND Rear = NULL THEN
PRINT “Queue is empty”
GOTO Step 8
[END IF]
Step 2: [INITIALIZE] ptr
Step 3: ptr = Front
Step 4: REPEAT Steps 6, 7 WHILE ptr -> Next != NULL
Step 5: PRINT ptr->data
Step 6: ptr = ptr -> Next
[END LOOP]
Step 7: PRINT ptr -> Data
Step 8: EXIT

## Implementation of Queue using Singly linked list

```
// code
#include <stdio.h>
#include <stdlib.h>
// Implementation of queue with liked list

// Declaration of node of linked list
struct node {
   int data;
   struct node *next;
};

// front of linked list
struct node *front = NULL;

// rear of linked list
struct node *rear = NULL;


void insert(int val) { // insert value at front
   // declare and allocate memory of newNode
   struct node *newNode;
   newNode = (struct node *)malloc(sizeof(struct node));

   newNode->data = val;

   if (front == NULL && rear == NULL) {    // when first node is added
      front = newNode;
      rear = newNode;
      newNode->next = NULL;
   }
   else { // insertion of any other node
      rear->next = newNode;
      rear = newNode;
      newNode->next = NULL;
   }
}

void delete () {    // deletes node at front
   // traversing pointer
   struct node *ptr;
   ptr = front;

   if (front == NULL && rear == NULL) {     // checks if queue is empty
      printf("\nQueue is empty!");
      return;
   }

   // moves 'front' ahead
   front = ptr->next;
   printf("\nDeleted element is :  %d", ptr->data);
```

```
    free(ptr);

    if (front == NULL) {    // when last node is deleted
        rear = NULL;
    }
}

void showFront() {    //displays element at front

    if (front == NULL && rear == NULL) {     // checks if queue is empty
        printf("\nQueue is empty!");
        return;
    }

    // displays element at front
    printf("\nElement at front is :  %d", front->data);
}

int size() {    // returns size of the queue

    if (front == NULL && rear == NULL) {     // if queue is empty
        return 0;
    }
    int count = 1;

    // traversing pointer
    struct node *ptr;
    ptr = front;
    while (ptr->next != NULL) {    // count number of nodes in queue
        ptr = ptr->next;
        count++;
    }
    return count;
}

void display() {    // display elements of queue
    // traversing pointer
    struct node *ptr;
    ptr = front;

    if (front == NULL && rear == NULL) {     // checks if queue is empty
        printf("\nQueue is empty!");
        return;
    }
    printf("\nElements in queue are :  ");
    while (ptr->next != NULL) {    // traverse and display
        printf("%d  ", ptr->data);
        ptr = ptr->next;
    }
    printf("%d", ptr->data);
}
```

```
int main() {

    int choice, val;

    while (1) {
        printf("\n*1. INSERT");
        printf("\n*2. DELETE");
        printf("\n*3. SHOW FRONT");
        printf("\n*4. SIZE");
        printf("\n*5. DISPLAY");
        printf("\n*6. EXIT");
        printf("\nEnter your choice :  ");
        scanf("%d", &choice);

        switch (choice) {

        case 1:
            printf("\nEnter an element to insert :  ");
            scanf("%d", &val);
            insert(val);
            break;
        case 2:
            delete();
            break;
        case 3:
            showFront();
            break;
        case 4:
            printf("\nSize of queue is :  %d", size());
            break;
        case 5:
            display();
            break;
        case 6:
            printf("\n *** E X I T I N G ***");
            exit(1);
        default:
            printf("\nINVALID INPUT");
        }
    }
    return 0;
}
```

// Output

```
*1. INSERT
*2. DELETE
*3. SHOW FRONT
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  1

Enter an element to insert :  5

*1. INSERT
*2. DELETE
*3. SHOW FRONT
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  1

Enter an element to insert :  10

*1. INSERT
*2. DELETE
*3. SHOW FRONT
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  1

Enter an element to insert :  15

*1. INSERT
*2. DELETE
*3. SHOW FRONT
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  4

Size of queue is :  3
*1. INSERT
*2. DELETE
*3. SHOW FRONT
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  5

Elements in queue are :  5  10  15
```

```
*1. INSERT
*2. DELETE
*3. SHOW FRONT
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  2

Deleted element is :  5
*1. INSERT
*2. DELETE
*3. SHOW FRONT
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  3

Element at front is :  10
*1. INSERT
*2. DELETE
*3. SHOW FRONT
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  2

Deleted element is :  10
*1. INSERT
*2. DELETE
*3. SHOW FRONT
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  5

Elements in queue are :  15
*1. INSERT
*2. DELETE
*3. SHOW FRONT
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  2

Deleted element is :  15
```

```
*1. INSERT
*2. DELETE
*3. SHOW FRONT
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  1

Enter an element to insert :  200

*1. INSERT
*2. DELETE
*3. SHOW FRONT
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  5

Elements in queue are :  200
*1. INSERT
*2. DELETE
*3. SHOW FRONT
*4. SIZE
*5. DISPLAY
*6. EXIT
Enter your choice :  6

 *** E X I T I N G ***
```

**DSA Write-up**
**Experiment number 09**

**Name:** Aamir Ansari **Batch:** A **Roll no.** 01

**AIM:** To implement of Singly Circular linked lists

## THEORY:

A circular linked list is the type of linked list in which the last node contains a pointer to the first node of the list. A circular linked list has no beginning and no ending.

## ALGORITHM

INSERT

**At the beginning**

```
Step 1: [INITIALIZE] newNode
Step 2: SET newNode->data = data
Step 3: IF end == NULL
                SET end = newNode
                SET newNode->next = end
                Goto Step 5
Step 4: ELSE
                newNode->next = end->next
                end->next = newNode
        [END IF]
Step 5: EXIT
```

**At the end**

```
Step 1: [INITIALIZE] newNode, ptr
Step 2: SET newNode->data = data
Step 3: IF end == NULL
                SET end = newNode
                SET newNode->next = end
                Goto Step 5
Step 4: ELSE
                SET ptr = end->next
                Repeat while ptr->next != end
                        ptr=ptr-> next
                [END LOOP]
                newNode->next = end->next
                end->next = newNode
                end = newNode
        [END IF]
Step 5: EXIT
```

**At a position:**

```
Step 1: [INITIALIZE] newNode, ptr, prePtr
Step 2: SET newNode = end->next->next , prenewNode = newNode
Step 3: SET new->data = data
Step 4: IF end == NULL
```

PRINT “LIST EMPTY”
Goto Step 12
[END IF]
Step 4: SET count = 1
Step 5:Repaet step 6 to 8 while count!=position AND ptr->next!=end->next
Step 6: SET prePtr = ptr;
Step 7: SET ptr = ptr->next;
Step 8: count = count + 1
Step 9: IF count == 1
newNode->next = ptr
end->next = newNode
Step 10: ELSE IF ptr->next == end->next AND count < pos
newNode->next = end->next
end->next = newNode
end = newNode
Step 11:ELSE
prePtr->next = newNode;
newNode->next = ptr;
[END IF]
Step 12:EXIT

**Before a given value:**

Step 1: [INITIALIZE] newNode, ptr, prePtr
Step 2: SET new->data= data
Step 3: SET newNode=end->next
Step 4: SET prePtr = ptr
Step 5: IF end == NULL
PRINT “LIST IS EMPTY”
Goto Step 11
Step 6: Repeat step 7&8 while newNode->data != val
Step 7: SET prePtr = ptr;
Step 8: SET ptr = ptr->next;
Step 9:IF ptr == end->next
SET newNode->next = end->next;
SET end->next = newNode;
Step 10: ELSE
SET prePtr->next = newNode
SET newNode->next = ptr
Step 11:EXIT

**After a given Value:**

Step 1: [INITIALIZE] ptr, prePtr, newNode
Step 2: IF end == NULL
PRINT “LIST IS EMPTY”
Goto Step 9
Step 3: SET ptr = end->next
Step 4: DO steps 5&6 while ptr->data != val
Step 5: SET prePtr = ptr;
Step 6: SET ptr = ptr->next;
[END LOOP]
Step 7: IF prePtr->next == end->next
newNode->next = end->next;

```
                prePtr->next = newNode;
                end = newNode;
Step 8: ELSE
                prePtr->next = newNode;
                newNode->next = ptr;
        [END IF]
Step 9:EXIT
```

DELETE

**Value at the beginning**

```
Step 1: [INITIALIZE] ptr
Step 2: IF end == NULL
                PRINT "LIST IS EMPTY"
                Goto Step 6
        [END IF]
Step 3: SET end->next == ptr->next
Step 4: IF ptr == end
                end = NULL
        [END IF]
Step 5: free(ptr)
Step 6: EXIT
```

**At the end**

```
Step 1: [INITIALIZE] ptr, prePtr
Step 2: IF end == NULL
                PRINT "LIST IS EMPTY"
                Goto Step 10
        [END IF]
Step 3: Repeat Steps 4, 5 while ptr->next != end->next
Step 4:         SET prePtr = ptr;
Step 5:         SET ptr = ptr->next;
        [END LOOP]
Step 6: SET prePtr->next = end->next;
Step 7: SET end = prePtr;
Step 8: IF prePtr == ptr
                SET end = NULL
        [END IF]
Step 9: free(ptr)
Step 10: EXIT
```

**Value at a Position**

```
Step 1: [INITIALIZE] ptr, prePtr
Step 2: IF end == NULL
                PRINT "LIST IS EMPTY"
                Goto Step 13
        [END IF]
Step 3: SET count = 1
Step 4: Repeat steps  while count != pos AND ptr->next != end->next
Step 5:         SET prePtr = ptr;
Step 6:         SET ptr = ptr->next;
```

```
Step 7:         SET count = count + 1;
        [END LOOP]
Step 8: IF POS > count
                PRINT “NO NODE AVAILABLE”
                Goto Step 13
        [END IF]
Step 9: IF end->next == ptr
                SET end->next = ptr->next;
                free(ptr);
Step 10: ELSE IF ptr->next == end->next
                SET prePtr->next = end->next;
                SET end = prePtr;
                SET end->next = prePtr->next;
                free(ptr);
Step 11: ELSE
                SET prePtr->next = ptr->next;
                free(ptr);
        [END IF]
Step 12: IF ptr->next == end->next
                SET end =NULL
        [END IF]
Step 13: EXIT
```

**Before a given value**

```
Step 1: [INITIALIZE] ptr, prePtr
Step 2: IF end == NULL
                PRINT "LIST IS EMPTY"
                Goto Step 9
        [END IF]
Step 3: IF ptr->data == val
                PRINT “NO NODE BEFORE THIS”
                Goto Step 9
        [END IF]
Step 4: Repeat Steps 5, 6 while ptr->next->data != val
Step 5:         SET prePtr = ptr;
Step 6:         SET ptr = ptr->next;
        [END LOOP]
Step 7: prePtr-next = ptr->next
Step 8: free(ptr)
Step 9: EXIT
```

**After a given value**

```
Step 1: [INITIALIZE] ptr, prePtr
Step 2: IF end == NULL
                PRINT "LIST IS EMPTY"
                Goto Step 10
        [END IF]
Step 3: Repeat Steps 5, 6 while ptr->data != val
Step 4:         SET prePtr = ptr;
Step 5:         SET ptr = ptr->next;
        [END LOOP]
Step 6: prePtr = ptr
```

Step 7: ptr = ptr->next
Step 8: IF ptr->next == end->next
SET prePtr->next = end->next;
SET end = prePtr;
free(ptr);
Step 9: ELSE
SET prePtr->next = ptr->next;
free(ptr);
[END IF]
Step 10: EXIT

3.UPDATE

**Value at the beginning**

Step 1: IF end == NULL
PRINT "LIST IS EMPTY"
Goto Step 3
[END IF]
Step 2: SET end->next->data = toUpdate
Step 3:EXIT

**At the end**

Step 1: IF end == NULL
PRINT "LIST IS EMPTY"
Goto Step 3
[END IF]
Step 2: SET end->data = toUpdate
Step 3:EXIT

**Value at a given Position**

Step 1: IF end == NULL
PRINT "LIST IS EMPTY"
Goto Step 9
[END IF]
Step 2: SET count = 1
Step 3: [INITIALIZE] ptr
Step 4: Repeat step 5, 6 while count != pos AND ptr->next!=end->next
Step 5: SET ptr = ptr->next
Step 6: SET count = count + 1
[END LOOP]
Step 7: IF pos > count
PRINT “NO NODE AT GIVEN POSITION”
Goto Step 9
[END IF]
Step 8: SET ptr->data = toUpdate
Step 9: EXIT

**Before a particular value**

Step 1: IF end == NULL
PRINT "LIST IS EMPTY"
Goto Step 6

[END IF]
Step 2: [INITIALIZE] ptr
Step 3:Repeat step 4&5 while ptr->next->data != val
Step 4: SET ptr = ptr->next
[END LOOP]
Step 5: SET ptr->data = toUpdate;
Step 6: EXIT

**After a particular value**

Step 1: IF end == NULL
PRINT "LIST IS EMPTY"
Goto Step 6
[END IF]
Step 2: [INITIALIZE] ptr
Step 3:Repeat step 4&5 while ptr->next->data != val
Step 4: SET ptr = ptr->next
[END LOOP]
Step 5: SET ptr->next->data = toUpdate;
Step 6: EXIT

4. SEARCH

Step 1: IF end == NULL
PRINT “LIST IS EMPTY”
Goto Step 9
[END IF]
Step 2: [INITIALIZE] ptr
Step 3: SET Count = 1
Step 4: Repeat step 4&5 while ptr->data != val && count<=countNodes()+1
Step 5: SET ptr = ptr->next
Step 6: SET count = count + 1
[END LOOP]
Step 7: IF count > countNodes()
PRINT “NOT FOUND”
Step 8: ELSE
PRINT “FOUND”
Step 9: EXIT

5. COUNT NODES

Step 1: IF end == NULL
return 0
[END IF]
Step 2: [INITIALIZE] ptr
Step 3: SET Count = 1
Step 4:Repeat step 5, 6 while ptr->next->data != val
Step 5: SET ptr = ptr->next
Step 6: SET count = count + 1
[END LOOP]
Step 7: return count

6. DISPLAY

Step 1: IF end == NULL
PRINT “LIST IS EMPTY”
Goto Step 7
[END IF]
Step 2: [INITIALIZE] ptr
Step 3: Repeat steps 4, 5 while ptr->next != end->next
Step 4: PRINT ptr->data
Step 5: ptr = ptr->next;
[END LOOP]
Step 6: PRINT ptr->data
Step 7: EXIT

**********

## Implementation of circular linked list

```
// code
#include <stdio.h>
#include <stdlib.h>
// Implementation of circular linked list

// declaration of node of linked list
struct node {
    int data;
    struct node *next;
};

// declaration of end
struct node *end = NULL;

void insertAtBegining(int toInsert) {    // Insert at the begining of list

    // declaring, inserting value and allocating memory for new node
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = toInsert;

    if (end == NULL) {    // if first node is to be added
        end = newNode;
        newNode->next = end;
    } else {    // inserting node at the begining
        newNode->next = end->next;
        end->next = newNode;
    }
}

void insertAtEnd(int toInsert) {    // Insert at the end of the list

    // declaring, inserting value and allocating memory for new node
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = toInsert;

    if (end == NULL) {    // if first node is to be added
        end = newNode;
        newNode->next = end;
    } else {    // inserting node at the end

        // traversing pointer
        struct node *ptr;
        ptr = end->next;

        while (ptr->next != end) {
            ptr = ptr->next;
        }
        newNode->next = end->next;
```

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

}

void insertBeforeVal(int toInsert, int val) {    // Insert before valule (val) is encountered

    // traversing pointer
    struct node *ptr;
    ptr = end->next;
    struct node *prePtr;
    prePtr = ptr;

    // declaring, inserting value and allocating memory for new node
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = toInsert;

    if (end == NULL) {    // check if list is empty
        printf("\nList is empty!");
        return;
    }

    // traversing upto val in the list
    while (ptr->data != val) {
        prePtr = ptr;
        ptr = ptr->next;
    }

    if (ptr == end->next) {    // adding before first node
        newNode->next = end->next;
        end->next = newNode;
    } else {    // adding before any nodes
        prePtr->next = newNode;
        newNode->next = ptr;
    }
}

void insertAfterVal(int toInsert, int val) {    // Inserts node after value (val) is encountered

    // traversing pointer
    struct node *ptr;
    ptr = end->next;
    struct node *prePtr;
    prePtr = ptr;

    // declaring, inserting value and allocating memory for new node
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = toInsert;
```

```
    if (end == NULL) {    // check if list is empty
        printf("\nList is empty!");
        return;
    }

    // traversing until val is encountered
    while (ptr->data != val) {
        prePtr = ptr;
        ptr = ptr->next;
    }
    prePtr = ptr;
    ptr = ptr->next;

    if (prePtr->next == end->next) {    // inserting node after last node
        newNode->next = end->next;
        prePtr->next = newNode;
        end = newNode;
    } else {    // inserting after any node
        prePtr->next = newNode;
        newNode->next = ptr;
    }

}

void insertAtPosition(int toInsert, int pos) {    // inserting after given position

    // traversing pointer
    struct node *ptr;
    ptr = end->next;
    struct node *prePtr;
    prePtr = ptr;

    // declaring, inserting value and allocating memory for new node
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = toInsert;

    if (end == NULL) {    // check if list is empty
        printf("\nList is empty!");
        return;
    }

    int count = 1;

    while (count!=pos && ptr->next!=end->next) {
        prePtr = ptr;
        ptr = ptr->next;
        count++;
    }

    if (pos > count+1) {    // invalid position
        printf("\nList is not that long!");
```

```
        return;
    }

    if (count == 1) {    // adding new node before first node
        newNode->next = ptr;
        end->next = newNode;
    } else if (ptr->next == end->next && count < pos) {    // inserting after last node  /* second
confition => when the postion is second-last */
        newNode->next = end->next;
        end->next = newNode;
        end = newNode;
    } else {    // inserting at any position
        prePtr->next = newNode;
        newNode->next = ptr;
    }

}

void deleteAtBeginning() {    // Deletes node at the beginning

    if (end == NULL) {    // checks if list is empty
        printf("\nList is empty!");
        return;
    }

    // traversing pointer
    struct node *ptr = end->next;
    printf("\nDeleted element is :  %d", ptr->data);

    // shiftind end->next to second node
    end->next = ptr->next;

    if (ptr == end) {    // when only remaining node is deleted
        end = NULL;
    }
    free(ptr);
}

void deleteAtEnd() {    // Deletes node at the end of the linked list

    if (end == NULL) {    // checks if list is empty
        printf("\nList is empty!");
        return;
    }

    // traversing pointer
    struct node *ptr = end->next;
    struct node *prePtr = ptr;

    while (ptr->next != end->next) {
        prePtr = ptr;
        ptr = ptr->next;
```

```
    }
    printf("\nDeleted element is :  %d", ptr->data);

    // shifting end to second-last node
    prePtr->next = end->next;
    end = prePtr;
    if (prePtr == ptr) {    // when only remaining node is deleted
        end = NULL;
    }
    free(ptr);
}

void deleteBeforeVal(int val) {    // Deletes node before given value (val)

    if (end == NULL) {    // checks if list is empty
        printf("\nList is empty!");
        return;
    }

    // traversing pointer
    struct node *ptr = end->next;
    struct node *prePtr = ptr;

    if (ptr->data == val) {    // if the val is of first node
        printf("\nThere is no node before this!");
        return;
    }

    // traversing
    while (ptr->next->data != val) {
        prePtr = ptr;
        ptr = ptr->next;
    }
    // deleting
    prePtr->next = ptr->next;
    free(ptr);
}

void deleteAfterVal(int val) {    // Deletes node after value (val) is encountered

    if (end == NULL) {    // checks if list is empty
        printf("\nList is empty!");
        return;
    }

    // traversing pointer
    struct node *ptr = end->next;
    struct node *prePtr = ptr;

    // traversing the list
    while (ptr->data != val) {
        prePtr = ptr;
```

```
        ptr = ptr->next;
    }
    prePtr = ptr;
    ptr = ptr->next;

    printf("\nDeleted element is :  %d", ptr->data);

    if (ptr->next == end->next) {    // last node is deleted
        prePtr->next = end->next;
        end = prePtr;
        free(ptr);
    } else {    // andy other node is deleted
        prePtr->next = ptr->next;
        free(ptr);
    }
}

void deleteAtPosition(int pos) {    // Deletes node at given position

    if (end == NULL) {    // checks if list is empty
        printf("\nList is empty!");
        return;
    }

    // traversing pointer
    struct node *ptr = end->next;
    struct node *prePtr = ptr;

    int count = 1;

    while (count!=pos && ptr->next!=end->next) {
        prePtr = ptr;
        ptr = ptr->next;
        count++;
    }

    if (pos > count) {    // invalid pos
        printf("\nThere is no node at this position");
        return;
    }

    printf("\nDeleted element is :  %d", ptr->data);

    if (end->next == ptr) {    // deleting at first position
        end->next = ptr->next;
        free(ptr);
    } else if (ptr->next == end->next) {    // deleting at last position
        prePtr->next = end->next;
        end = prePtr;
        end->next = prePtr->next;
        free(ptr);
    } else {    // deleting at any position
```

```
        prePtr->next = ptr->next;
        free(ptr);
    }

    if (ptr->next == end->next) {    // only remaining node is deleted
        end = NULL;
    }

}

void updateAtBeginning(int toUpdate) {    // Updates the element at the beginning

    if (end == NULL) {    // check if the list is empty
        printf("\nList is Empty!");
        return;
    }

    // updation
    end->next->data = toUpdate;

}

void updateAtEnd(int toUpdate) {    // Updates the element at the end

    if (end == NULL) {    // check if the list is empty
        printf("\nList is Empty!");
        return;
    }

    // updation
    end->data = toUpdate;

}

void updateBeforeVal(int toUpdate, int val) {    // Updates element before a given val

    if (end == NULL) {    // check if the list is empty
        printf("\nList is Empty!");
        return;
    }

    if (end->next->data == val) {    // if the value is of first node
        printf("\nThere are no elements before this!");
        return;
    }

    // traversing pointer
    struct node *ptr = end->next;

    // traversing
    while (ptr->next->data != val) {
        ptr = ptr->next;
```

```
    }

    // updation
    ptr->data = toUpdate;
}

void updateAfterVal(int toUpdate, int val) {    // Updates the element after value (val) is
encountered

    if (end == NULL) {    // check if the list is empty
        printf("\nList is Empty!");
        return;
    }

    // traversing pointer
    struct node *ptr = end->next;

    // traversing
    while (ptr->data != val) {
        ptr = ptr->next;
    }

    // updation
    ptr->next->data = toUpdate;
}

void updateAtPosition(int toUpdate, int pos) {    // Updates value at given position

    if (end == NULL) {    // check if the list is empty
        printf("\nList is Empty!");
        return;
    }

    // traversing pointer
    struct node *ptr = end->next;

    int count = 1;

    // traversing
    while (count != pos && ptr->next!=end->next) {
        ptr = ptr->next;
        count++;
    }

    if (pos > count) {    // checks for valid position
        printf("\nNo node at the given position!");
        return;
    }

    // updation
    ptr->data = toUpdate;
```

```
}

int countNodes() {    // Counts number of nodes in the list

    if (end == NULL) {    // if the list is empty
        return 0;
    }

    // traversing pointer
    struct node *ptr = end->next;

    int count = 1;

    // traversing
    while (ptr->next!=end->next) {
        ptr = ptr->next;
        count++;
    }

    return count;
}

void search(int val) {    // Search weather the val is present in the list and prints its position

    if (end == NULL) {    // check if the list is empty
        printf("\nList is Empty!");
        return;
    }

    // traversing pointer
    struct node *ptr = end->next;

    int count = 1;

    // traversing
    while (ptr->data != val && count<=countNodes()+1) {
        ptr = ptr->next;
        count++;
    }

    // printing
    if (count > countNodes()) {
        printf("\n%d is not present in the list!", val);
    } else {
        printf("\nPosition of %d in the list is :  %d", val, count);
    }

}

void display() {    // Displays content of linked list

    // traversing pointer
```

```
    struct node *ptr;

    if (end == NULL) {    // check if list is empty
        printf("\nList is empty!");
        return;
    }
    // initializing traversing pointer
    ptr = end->next;

    // printing
    while (ptr->next != end->next) {
        printf("%d,  ", ptr->data);
        ptr = ptr->next;
    }
    printf("%d  ", ptr->data);

}

int main() {

    int choice, toInsert, toUpdate, val, pos;

    while (1) {

        printf("\n*1   INSERT At END  ");
        printf("\n*2   INSERT At BEGINING  ");
        printf("\n*3   INSERT BEFORE VAL  ");
        printf("\n*4   INSERT AFTER VAL  ");
        printf("\n*5   INSERT At POSITION  ");
        printf("\n*6   DELETE At END  ");
        printf("\n*7   DELETE At BEGINING  ");
        printf("\n*8   DELETE BEFORE VAL  ");
        printf("\n*9   DELETE AFTER VAL  ");
        printf("\n*10  DELETE At POSITION  ");
        printf("\n*11  UPDATE At END  ");
        printf("\n*12  UPDATE At BEGINING  ");
        printf("\n*13  UPDATE BEFORE VAL  ");
        printf("\n*14  UPDATE AFTER VAL  ");
        printf("\n*15  UPDATE At POSITION  ");
        printf("\n*16  SEARCH in the list  ");
        printf("\n*17  COUNT NODE in the list  ");
        printf("\n*18  DISPLAY elements of the list  ");
        printf("\n*19  EXIT  ");
        printf("\nEnter your choice :  ");
        scanf("%d", &choice);

        switch (choice) {

            case 1:
                printf("\nEnter element to insert :  ");
                scanf("%d", &toInsert);
                insertAtEnd(toInsert);
```

```
    break;

case 2:
    printf("\nEnter element to insert :  ");
    scanf("%d", &toInsert);
    insertAtBegining(toInsert);
    break;

case 3:
    printf("\nEnter element to insert :  ");
    scanf("%d", &toInsert);
    printf("\nEnter value BEFORE which to insert :  ");
    scanf("%d", &val);
    insertBeforeVal(toInsert, val);
    break;

case 4:
    printf("\nEnter element to insert :  ");
    scanf("%d", &toInsert);
    printf("\nEnter value AFTER which to insert :  ");
    scanf("%d", &val);
    insertAfterVal(toInsert, val);
    break;

case 5:
    printf("\nEnter element to insert :  ");
    scanf("%d", &toInsert);
    printf("\nEnter POSITION AT which to insert :  ");
    scanf("%d", &pos);
    insertAtPosition(toInsert, pos);
    break;

case 6:
    deleteAtEnd();
    break;

case 7:
    deleteAtBeginning();
    break;

case 8:
    printf("\nEnter value BEFORE which to DELETE :  ");
    scanf("%d", &val);
    deleteBeforeVal(val);
    break;

case 9:
    printf("\nEnter value AFTER which to DELETE :  ");
    scanf("%d", &val);
    deleteAfterVal(val);
    break;
```

```
case 10:
    printf("\nEnter POSITION AT which to DELETE :  ");
    scanf("%d", &pos);
    deleteAtPosition(pos);
    break;

case 11:
    printf("\nEnter element to UPDATE :  ");
    scanf("%d", &toUpdate);
    updateAtEnd(toUpdate);
    break;

case 12:
    printf("\nEnter element to UPDATE :  ");
    scanf("%d", &toUpdate);
    updateAtBeginning(toUpdate);
    break;

case 13:
    printf("\nEnter element to UPDATE :  ");
    scanf("%d", &toUpdate);
    printf("\nEnter value BEFORE which to UPDATE :  ");
    scanf("%d", &val);
    updateBeforeVal(toUpdate, val);
    break;

case 14:
    printf("\nEnter element to UPDATE :  ");
    scanf("%d", &toUpdate);
    printf("\nEnter value AFTER which to UPDATE :  ");
    scanf("%d", &val);
    updateBeforeVal(toUpdate, val);
    break;

case 15:
    printf("\nEnter element to UPDATE :  ");
    scanf("%d", &toUpdate);
    printf("\nEnter POSITION AT which to UPDATE :  ");
    scanf("%d", &pos);
    updateAtPosition(toUpdate, pos);
    break;

case 16:
    printf("\nEnter a value to SEARCH :  ");
    scanf("%d", &val);
    search(val);
    break;

case 17:
    printf("\nList contains  %d  elements", countNodes());
    break;
```

```
        case 18:
            printf("\nElements in the list are :  ");
            display();
            break;

        case 19:
            printf("*** E X I T I N G ***");
            exit(1);
            break;

        default:
            printf("INVALID INPUT");
      }

   }

   return 0;

}
```

// output

```
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  EXIT
Enter your choice :  1

Enter element to insert :  5

*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  EXIT
Enter your choice :  1

Enter element to insert :  10
```

```
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  EXIT
Enter your choice :  1

Enter element to insert :  15

*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  EXIT
Enter your choice :  1

Enter element to insert :  20
```

```
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  EXIT
Enter your choice :  18

Elements in the list are :  5,  10,  15,  20
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  EXIT
Enter your choice :  17

List contains  4  elements
```

```
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  EXIT
Enter your choice :  16

Enter a value to SEARCH :  10

Position of 10 in the list is :  2
List contains  4  elements
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  EXIT
Enter your choice :  6

Deleted element is :  20
```

```
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  EXIT
Enter your choice :  7

Deleted element is :  5
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  EXIT
Enter your choice :  18

Elements in the list are :  10,  15
```

```
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  EXIT
Enter your choice :  19
*** E X I T I N G ***
Process returned 1 (0x1)   execution time : 120.321 s
Press any key to continue.
```

**DSA LAB**
**Experiment number 10**

**Name:** Aamir Ansari **Batch:** A **Roll no.** 01

**Aim:** Implementation of Josephus Problem using Circular LL.

## Theory:

There are n people standing in a circle waiting to be executed. The counting out begins at some point in the circle and proceeds around the circle in a fixed direction. In each step, a certain number of people are skipped and the next person is executed. The elimination proceeds around the circle (which is becoming smaller and smaller as the executed people are removed), until only the last person remains, who is given freedom. Given the total number of persons n and a number k which indicates that k-1 persons are skipped and $k^{th}$ person is killed in circle. The task is to choose the place in the initial circle so that you are the last one remaining and so survive.

For example, if n = 5 and k = 2, then the safe position is 3. Firstly, the person at position 2 is killed, then person at position 4 is killed, then person at position 1 is killed. Finally, the person at position 5 is killed. So the person at position 3 survives.

## Algorithm:

Step 1: START
Step 2: Create circular singly linked list with id and next pointer as elements of node
Step 3: Set id of node same as the position of node
Step 4: Repeat steps 5, 6 while there is more than one node in the list
Step 5: move forward k nodes in the list
Step 6: delete the node
Step 7: Print the id of only remaining node as winner
Step 8: EXIT

## Implementation of Josephus Problem using Circular LL

```
// code:
#include <stdio.h>
#include <stdlib.h>
// Josephus problem using circular linked list
struct node {
   int id;
   struct node *next;
};

struct node *start, *ptr, *newNode;

int main() {
   int i, n, k, count;
   printf("Enter the number of player :  ");
   scanf("%d", &n);
   printf("Enter value of k :  ");
   scanf("%d", &k);

   start = malloc(sizeof(struct node));
   start->id = 1;
   ptr = start;
   // creating list of size n
   for (i = 2 ; i <= n ; i++) {
      newNode = malloc(sizeof(struct node));
      ptr->next = newNode;
      newNode->id = i;
      newNode->next = start;
      ptr = newNode;
   }
   // traversing the list and deleting every kth node
   for (count = n ; count>1 ; count--) {

      for (i = 0 ; i < k-1 ; i++) {
         ptr = ptr->next;
      }
      ptr->next = ptr->next->next;
   }
   printf("\nW I N N E R :  %d", ptr->id);
}
```

// output

```
Enter the number of player :  5
Enter value of k :  2

W I N N E R :  3
Process returned 0 (0x0)   execution time : 3.743 s
Press any key to continue.
```

```
Enter the number of player :  7
Enter value of k :  3

W I N N E R :  4
Process returned 0 (0x0)   execution time : 4.812 s
Press any key to continue.
```

**DSA LAB**
**Lab Assignment number 11**

**Name:** Aamir Ansari **Batch:** A **Roll no:** 01

## **Aim:** Implementation of Doubly linked list

**Theory:**

Doubly linked list?

A Doubly Linked List (DLL) contains an extra pointer, typically called previous pointer, together with next pointer and data which are there in singly linked list

Advantages over singly linked list

1) A DLL can be traversed in both forward and backward direction.
2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.
3) We can quickly insert a new node before a given node.

In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

## Algorithms:

INSERT

**At the beginning**

```
Step 1: [INITIALIZE] newNode
Step 2: SET newNode->data = data
Step 3: IF start == NULL
                SET newNode->next = NULL;
                SET newNode->previous = NULL;
                SET start = newNode;
Step 4: ELSE
                SET newNode->next = start;
                SET newNode->previous = NULL;
                SET start->previous = newNode;
                SET start = newNode;
        [END IF]
Step 5: EXIT
```

**At the end**

```
Step 1: [INITIALIZE] newNode, ptr
Step 2: SET newNode->data = data
Step 3: IF start == NULL
                SET newNode->next = NULL;
                SET newNode->previous = NULL;
```

```
                SET start = newNode;
Step 4: ELSE
                SET ptr = end->next
Repeat while ptr->next != end
        ptr=ptr-> next
[END LOOP]
        SET ptr->next = newNode;
        SET newNode->previous = ptr;
        SET newNode->next = NULL;
[END IF]
Step 5: EXIT
```

**At a position:**

```
Step 1: [INITIALIZE] newNode, ptr
Step 2: SET newNode = start
Step 3: SET new->data = data
Step 4: IF start == NULL
                PRINT "LIST EMPTY"
                Goto Step 12
        [END IF]
Step 4: SET count = 1
Step 5:Repaet step 6 to 8 while count!=position AND ptr->next!=end->next
Step 6:         SET prePtr = ptr;
Step 7:         SET ptr = ptr->next;
Step 8:         count = count + 1
Step 9: IF count == 1
                SET newNode->next = ptr;
                SET newNode->previous = NULL;
                SET ptr->previous = newNode;
                SET start = newNode;
Step 10: ELSE IF  ptr->next == end->next AND count < pos
                SET ptr->next = newNode;
                SET newNode->previous = ptr;
                SET newNode->next = NULL;
Step 11:ELSE
                SET newNode->next = ptr;
                SET newNode->previous = ptr->previous;
                SET ptr->previous->next = newNode;
                SET ptr->previous = newNode;
[END IF]
Step 12:EXIT
```

**Before a given value:**

```
Step 1: [INITIALIZE] newNode, ptr
Step 2: SET newNode->data= data
Step 3: SET ptr=start
Step 4: IF end == NULL
                PRINT "LIST IS EMPTY"
                Goto Step 9
Step 5: Repeat step 6&7 while newNode->data != val
Step 6:         SET ptr = ptr->next;
Step 7:IF ptr->previous == NULL
```

SET newNode->next = ptr;
SET newNode->previous = NULL;
SET ptr->previous = newNode;
SET start = newNode;
Step 8: ELSE
SET newNode->next = ptr;
SET newNode->previous = ptr->previous;
SET ptr->previous->next = newNode;
SET ptr->previous = newNode;
Step 9:EXIT

**After a given Value:**
Step 1: [INITIALIZE] newNode, ptr
Step 2: SET newNode->data= data
Step 3: SET ptr=start
Step 4: IF end == NULL
PRINT “LIST IS EMPTY”
Goto Step 10
Step 5: Repeat step 6&7 while ptr->data != val
Step 6: SET ptr = ptr->next;
Step 7: IF ptr->next == NULL
SET ptr->next = newNode;
SET newNode->previous = ptr;
SET newNode->next = NULL;
Step 8: ELSE
SET newNode->previous = ptr;
SET newNode->next = ptr->next;
SET ptr->next->previous = newNode;
SET ptr->next = newNode;
Step 9: EXIT

DELETE

**Value at the beginning**
Step 1: [INITIALIZE] ptr
Step 2: IF end == NULL
PRINT "LIST IS EMPTY"
Goto Step 6
[END IF]
Step 3: SET ptr = start
Step 4: IF ptr->next == NULL
SET start = NULL
Step 5: ELSE
SET ptr->next->previous = NULL;
SET start = ptr->next;
[END IF]
Step 5: free(ptr)
Step 6: EXIT

**At the end**

```
Step 1: [INITIALIZE] ptr
Step 2: IF start == NULL
                PRINT "LIST IS EMPTY"
                Goto Step 8
        [END IF]
Step 3: Repeat Steps 4, 5 while ptr->next != NULL
Step 4:         SET ptr = ptr->next;
        [END LOOP]
Step 5: IF start->next == NULL
                SET start = NULL
Step 6: ELSE
                SET ptr->previous->next = NULL;
        [END IF]
Step 7: free(ptr)
Step 8: EXIT
```

**Value at a Position**

```
Step 1: [INITIALIZE] ptr
Step 2: IF start == NULL
                PRINT "LIST IS EMPTY"
                Goto Step 12
[END IF]
Step 3: SET count = 1
Step 4: Repeat steps 5, 6 while count != pos AND ptr->next != NULL
Step 5:         SET ptr = ptr->next;
Step 6:         SET count = count + 1;
        [END LOOP]
Step 7: IF POS > count OR pos <= 0
                PRINT “NO NODE AVAILABLE”
                Goto Step 12
        [END IF]
Step 8: IF start->next == NULL
                SET start  = NULL
Step 9: ELSE IF count == 1
                SET ptr->next->previous = NULL;
                SET start = ptr->next;
Step 10: ELSE IF ptr->next == NULL
                SET ptr->previous->next = NULL
Step 11: ELSE
                SET ptr->previous->next = ptr->next;
                SET ptr->next->previous = ptr->previous;
        [END IF]
Step 12: EXIT
```

**Before a given value**

```
Step 1: [INITIALIZE] ptr
Step 2: IF start == NULL
                PRINT "LIST IS EMPTY"
                Goto Step 9
        [END IF]
Step 3: IF start->data == val
```

```
                PRINT “NO NODE BEFORE THIS”
                Goto Step 9
        [END IF]
Step 4: Repeat Step 5 while ptr->next->data != val
Step 5:         SET ptr = ptr->next;
        [END LOOP]
Step 6: IF ptr->previous == NULL THEN
                SET ptr->next->previous = NULL;
                SET start = ptr->next;
Step 7: ELSE
                SET ptr->previous->next = ptr->next;
                SET ptr->next->previous = ptr->previous;
        [END IF]
Step 8: free(ptr)
Step 9: EXIT
```

**After a given value**

```
Step 1: [INITIALIZE] ptr
Step 2: IF start == NULL
                PRINT "LIST IS EMPTY"
                Goto Step 10
        [END IF]
Step 3: Repeat Step 4 while ptr->data != val
Step 4:         SET ptr = ptr->next;
        [END LOOP]
Step 5: IF ptr->next == NULL THEN
                PRINT “NO ELEMENT AFTER THIS”
        [END IF]
Step 6: ptr = ptr->next
Step 7: IF ptr->next == NULL
                SET ptr->previous->next = NULL;
Step 8: ELSE
                SET ptr->previous->next = ptr->next;
                SET ptr->next->previous = ptr->previous;
        [END IF]
Step 9: free(ptr)
Step 10: EXIT
```

3.UPDATE

**Value at the beginning**

```
Step 1: IF start == NULL
                PRINT "LIST IS EMPTY"
                Goto Step 3
        [END IF]
Step 2: SET start->data = toUpdate;
Step 3:EXIT
```

**At the end**

```
Step 1: IF start == NULL
                PRINT "LIST IS EMPTY"
                Goto Step 5
        [END IF]
Step 2: Repeat Step 3 while ptr->data !=   NULL
Step 3:         SET ptr = ptr->next;
        [END LOOP]
Step 4: SET ptr->data = toUpdate;
Step 5:EXIT
```

**Value at a given Position**

```
Step 1: IF start == NULL
                PRINT "LIST IS EMPTY"
                Goto Step 9
        [END IF]
Step 2: SET count = 1
Step 3: [INITIALIZE] ptr
Step 4: Repeat step 5, 6 while count != pos AND ptr->next!= NULL THEN
Step 5:         SET ptr = ptr->next
Step 6:         SET count = count + 1
        [END LOOP]
Step 7: IF pos > count OR pos<=0 THEN
                PRINT “NO NODE AT GIVEN POSITION”
                Goto Step 9
        [END IF]
Step 8: SET ptr->data = toUpdate
Step 9: EXIT
```

**Before a particular value**

```
Step 1: IF start == NULL
                PRINT "LIST IS EMPTY"
                Goto Step 7
        [END IF]
Step 2: [INITIALIZE] ptr
Step 3: IF start->data == val  THEN
                PRINT “NO NODE BEFORE THIS”
Step 4: Repeat step 5 while ptr->next->data != val
Step 5:         SET ptr = ptr->next
        [END LOOP]
Step 6: SET ptr->data = toUpdate;
Step 7: EXIT
```

**After a particular value**

```
Step 1: IF end == NULL
                PRINT "LIST IS EMPTY"
                Goto Step 8
        [END IF]
Step 2: [INITIALIZE] ptr
Step 3:Repeat step 4&5 while ptr->next->data != val
Step 4:         SET ptr = ptr->next
        [END LOOP]
```

Step 5: IF ptr->next == NULL  THEN
PRINT “NO NODE AFTER THIS”
Step 6: SET ptr = ptr->next
Step 7: SET ptr->data = toUpdate;
Step 8: EXIT

4. COUNT NODES
Step 1: INITIALIZE count = 0, pr = START
Step 2: Repeat step 3&4 ptr->next != NULL
Step 3: SET count = count + 1
Step 4: SET ptr = ptr->next
Step 5: RETURN count
Step 6: EXIT

5. SEARCH
Step 1: SET PTR = START
Step 2: Repeat Step 3 while PTR != NULL
Step 3: IF VAL = PTR->DATA
PRINT ‘ELEMENT FOUND’
Go To Step 5
Step 4: ELSE
SET PTR = PTR->NEXT
[END OF IF]
Step 5: PRINT ‘ELEMENT NOT FOUND’
Step 6: EXIT

6. SORT
Step 1:[INITIALIZE] node traverse , min , temp
Step 2:Repeat step 3&4 while START->next
Step 3: SET min = START
Step 4: SET traverse = START->next
Step 5:Repeat step 6&7 while traverse is true
Step 6: IF min->data > traverse->data
SET min = traverse
Step 7: SET traverse = traverse->next
Step 8: SET temp = START->data
Step 9: SET START->data = min->data
Step 10: SET min->data = temp
Step 11: SET START = START->next
Step 12:EXIT

7. REVERSE
Step 1:[INITIALIZE] prev, ptr, next

Step 2:SET prev=NULL

Step 3:SET ptr=START

Step 4: Repeat step 5 to 8 while ptr!=NULL

Step 5: SET  next=ptr->next

Step 6: SET ptr->next=prev

Step 7: SET prev=ptr

Step 8: SET ptr=next

Step 9:SET START=prev

8. DISPLAY

Step 1: [INITIALIZE] ptr
Step 2: IF start == NULL
PRINT "LIST IS EMPTY"
Goto Step 7
[END IF]
Step 3: Repeat Step 4, 5 while ptr->data != val
Step 4: SET ptr = ptr->next;
Step 5: PRINT ptr->data
[END LOOP]
Step 6: PRINT ptr->data
Step 7: EXIT

## Implementation of Doubly linked list

```
// code
#include <stdio.h>
#include <stdlib.h>
// Implementation of Doubly linked list

//Declaration of node
struct node {
   int data;
   struct node *previous;
   struct node *next;
};

// Declararion of start of linked list
struct node *start = NULL;

// Second linked list for merging
struct nodeTwo {    // Declaration for secondary linked list
   int dataTwo;
   struct nodeTwo *previousTwo;
   struct nodeTwo *nextTwo;
};
// Start node of secondary linked list
struct nodeTwo *startTwo = NULL;

void secondLinkedList() {     // Initialises second linked list with static values
   // declare nodes
   struct nodeTwo *newNodeOne;
   struct nodeTwo *newNodeTwo;
   struct nodeTwo *newNodeThree;
   // allocates memory for nodes
   newNodeOne = (struct nodeTwo *)malloc(sizeof(struct nodeTwo));
   newNodeTwo = (struct nodeTwo *)malloc(sizeof(struct nodeTwo));
   newNodeThree = (struct nodeTwo *)malloc(sizeof(struct nodeTwo));
   // enter data and link the nodes
   startTwo = newNodeOne;
   newNodeOne->dataTwo = 4;
   newNodeOne->nextTwo = newNodeTwo;
   newNodeOne->previousTwo = NULL;

   newNodeTwo->dataTwo = 8;
   newNodeTwo->nextTwo = newNodeThree;
   newNodeTwo->previousTwo = newNodeOne;

   newNodeThree->dataTwo = 12;
   newNodeThree->nextTwo = NULL;
   newNodeThree->previousTwo = newNodeTwo;

}

void insertAtBeginning(int toInsert) {    // Inserts at the beginning of the node
```

```
    // declaration, memory allocation and initialization of new node
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = toInsert;

    if (start == NULL) {    // first node of is added
        newNode->next = NULL;
        newNode->previous = NULL;
        start = newNode;
    } else {
        // linking newNode before current start
        newNode->next = start;
        newNode->previous = NULL;
        start->previous = newNode;

        // shifting start
        start = newNode;
    }
}

void insertAtEnd(int toInsert) {    // Inserts at the end of the list

    // declaration, memory allocation and initialization of new node
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = toInsert;

    // traversing pointer
    struct node *ptr = start;

    if (start == NULL) {    // first node is to be added
        newNode->next = NULL;
        newNode->previous = NULL;
        start = newNode;
    } else {    // any other node
        while (ptr->next != NULL) {    // traverse upto currnet last node
            ptr = ptr->next;
        }
        // link current last node with newNode
        ptr->next = newNode;
        newNode->previous = ptr;
        newNode->next = NULL;
    }
}

void insertBeforeVal(int toInsert, int val) {    // Inserts node before val is encountered

    if (start == NULL) {    // checks if list is empty
        printf("\nList is empty!");
        return;
    }
```

```
    // declaration, memory allocation and initialization of new node
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = toInsert;

    // traversing pointer
    struct node *ptr = start;

    while (ptr->data != val) {    // traverse until val is encountered
        ptr = ptr->next;
    }

    if (ptr->previous == NULL) {    // inserting before current first node
        // linking new node with current first node
        newNode->next = ptr;
        newNode->previous = NULL;
        ptr->previous = newNode;
        // shifting start
        start = newNode;
    } else {
        // linking nodes before val
        newNode->next = ptr;
        newNode->previous = ptr->previous;
        ptr->previous->next = newNode;
        ptr->previous = newNode;
    }

}

void insertAfterVal(int toInsert, int val) {    // Inserts node after val is encountered

    if (start == NULL) {    // checks if list is empty
        printf("\nList is empty!");
        return;
    }

    // declaration, memory allocation and initialization of new node
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = toInsert;

    // traversing pointer
    struct node *ptr = start;

    while (ptr->data != val) {    // traverse until val is encountered
        ptr = ptr->next;
    }

    if (ptr->next == NULL) {    // inserting after current last node
        // linking new node with current last
        ptr->next = newNode;
```

```
        newNode->previous = ptr;
        newNode->next = NULL;
    } else {
        // linking nodes
        newNode->previous = ptr;
        newNode->next = ptr->next;
        ptr->next->previous = newNode;
        ptr->next = newNode;
    }

}

void insertAtPosition(int toInsert, int pos) {    // Inserts node at the given position

    if (start == NULL) {    // check if list is empty
        printf("\nList is empty!");
        return;
    }
    // declaration, memory allocation and initialization of new node
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = toInsert;

    // traversing pointer
    struct node *ptr = start;

    int count = 1;

    while (count != pos && ptr->next != NULL) {    // traverse list upto position
        ptr = ptr->next;
        count++;
    }

    if (pos > count+1 || pos <= 0) {    // invalid position
        printf("\nList is not that long!");
        return;
    }

    if (count == 1) {    // inserting at first position
        // linking new node with current first node
        newNode->next = ptr;
        newNode->previous = NULL;
        ptr->previous = newNode;
        // shifting start
        start = newNode;
    } else if (ptr->next == NULL && count < pos) {    // inserting at last position
        // linking new node with current last node
        ptr->next = newNode;
        newNode->previous = ptr;
        newNode->next = NULL;
    } else {    // inserting at any position
        newNode->next = ptr;
```

```
        newNode->previous = ptr->previous;
        ptr->previous->next = newNode;
        ptr->previous = newNode;
    }

}

void deleteAtBeginning() {    // deletes at the beginning

    if (start == NULL) {    // check if list is empty
        printf("\nList is empty!");
        return;
    }
    // traversing pointer
    struct node *ptr = start;
    printf("\nDeleted element is :  %d", ptr->data);

    if (ptr->next == NULL) {    // only remaining node is to be deleted
        start = NULL;
    } else {
        ptr->next->previous = NULL;
        start = ptr->next;
    }
    free(ptr);

}

void deleteAtEnd() {    // deletes at end

    if (start == NULL) {    // check if list is empty
        printf("\nList is empty!");
        return;
    }
    // traversing pointer
    struct node *ptr = start;

    while (ptr->next != NULL) {    // traversing upto last node
        ptr = ptr->next;
    }
    printf("\nDeleted element is :  %d", ptr->data);

    if (start->next == NULL) {    // only remaining node is to be deleted
        start = NULL;
    } else {
        ptr->previous->next = NULL;
    }
    free(ptr);

}

void deleteBeforeVal(int val) {    // deletes node before val is encountered
```

```
    if (start == NULL) {    // check if list is empty
        printf("\nList is empty!");
        return;
    }
    if (start->data == val) {    // check for invalid input
        printf("\nNo elements before  %d", val);
        return;
    }
    // traversing pointer
    struct node *ptr = start;

    while (ptr->next->data != val) {    // traversing upto last node
        ptr = ptr->next;
    }
    printf("\nDeleted element is :  %d", ptr->data);

    if (ptr->previous == NULL) {    // deleting current first node
        ptr->next->previous = NULL;
        start = ptr->next;
    } else {
        ptr->previous->next = ptr->next;
        ptr->next->previous = ptr->previous;
    }
    free(ptr);

}

void deleteAfterVal(int val) {    // deletes node after val is encountered

    if (start == NULL) {    // check if list is empty
        printf("\nList is empty!");
        return;
    }
    // traversing pointer
    struct node *ptr = start;

    while (ptr->data != val) {    // traversing until val is encountered
        ptr = ptr->next;
    }
    if (ptr->next == NULL) {    // check for invalid input
        printf("\nNo elements after  %d", val);
        return;
    }
    // set ptr to node which is to be deleted
    ptr = ptr->next;

    printf("\nDeleted element is :  %d", ptr->data);

    if (ptr->next == NULL) {    // deleting current last node
        ptr->previous->next = NULL;
    } else {
        ptr->previous->next = ptr->next;
```

```
        ptr->next->previous = ptr->previous;
    }
    free(ptr);

}

void deleteAtPosition(int pos) {    // deletes at given position

    if (start == NULL) {    // check if list is empty
        printf("\nList is empty!");
        return;
    }
    // traversing node
    struct node *ptr = start;
    int count = 1;

    while (count != pos && ptr->next != NULL) {    // traversing until val is encountered
        ptr = ptr->next;
        count++;
    }
    if (pos > count || pos<=0) {    // invalid position
        printf("\nInvalid position!");
        return;
    }
    printf("\nDeleted element is :  %d", ptr->data);

    if (start->next == NULL) {    // deleting only remaining node
       start = NULL;
    } else if (count == 1) {    // deleting at first position
        ptr->next->previous = NULL;
        start = ptr->next;
    } else if (ptr->next == NULL) {    // deleting at last position
        ptr->previous->next = NULL;
    } else {    // deleting at any position
        ptr->previous->next = ptr->next;
        ptr->next->previous = ptr->previous;
    }

}

void updateAtBeginning(int toUpdate) {    // updates at the beginning of the list

    if (start == NULL) {
        printf("\nList is empty!");
        return;
    }
    // updation of value
    start->data = toUpdate;
}

void updateAtEnd(int toUpdate) {    // updates at the end of the list
```

```
    if (start == NULL) {
        printf("\nList is empty!");
        return;
    }
    // traversing pointer
    struct node *ptr = start;
    while (ptr->next != NULL) {
        ptr = ptr->next;
    }
    // updation of value
    ptr->data = toUpdate;
}

void updateBeforeVal(int toUpdate, int val) {    // updates the node before val

    if (start == NULL) {
        printf("\nList is empty!");
        return;
    }
    if (start->data == val) {
        printf("\nNo nodes before entered value");
        return;
    }
    // traversing pointer
    struct node *ptr = start;
    while (ptr->next->data != val) {
        ptr = ptr->next;
    }
    // updation of value;
    ptr->data = toUpdate;
}

void updateAfterVal(int toUpdate, int val) {    // updates the node after val

    if (start == NULL) {
        printf("\nList is empty!");
        return;
    }
    // traversing pointer
    struct node *ptr = start;
    while (ptr->data != val) {
        ptr = ptr->next;
    }
    if (ptr->next == NULL) {
        printf("\nNo nodes after entered val!");
        return;
    }
    // shifting the pointer to node which is to be updated
    ptr = ptr->next;
    //updation
    ptr->data = toUpdate;
}
```

```
void updateAtPosition(int toUpdate, int pos) {    // updates value at enetered position

    if (start == NULL) {
        printf("\nList is empty!");
        return;
    }
    // traversing pointer
    struct node *ptr = start;
    int count = 1;

    while (count != pos && ptr->next != NULL) {
        ptr = ptr->next;
        count++;
    }
    if (pos > count || pos<=0) {    // invalid position
        printf("\nInvalid position!");
        return;
    }
    // updation
    ptr->data = toUpdate;
}

int countNodes() {    // Counts number of nodes in the list

    if (start == NULL) {    // if the list is empty
        return 0;
    }
    // traversing pointer
    struct node *ptr = start;
    int count = 1;

    // traversing
    while (ptr->next != NULL) {
        ptr = ptr->next;
        count++;
    }
    return count;

}

void search(int val) {    // Search weather the val is present in the list and prints its position

    if (start == NULL) {    // check if the list is empty
        printf("\nList is Empty!");
        return;
    }
    // traversing pointer
    struct node *ptr = start;
    int count = 1;

    // traversing
```

```
    while ((count!=countNodes()+1) && (ptr->data != val)) {
        ptr = ptr->next;
        count++;
    }
    // printing
    if (count > countNodes()) {
        printf("\n%d is not present in the list!", val);
    } else {
        printf("\nPosition of %d in the list is :  %d", val, count);
    }
}

void sort() {    // Sorts the list

    if (start == NULL) {    // check if the list is empty
        printf("\nList is Empty!");
        return;
    }
    struct node *i = start;
    struct node *j = NULL;
    int temp;
    for (i = start ; i != NULL ;  i=i->next) {
        for (j = i->next ; j != NULL ; j = j->next) {
            if (i->data > j->data) {
                temp = i->data;
                i->data = j->data;
                j->data = temp;
            }
        }
    }
}

void reverse() {    // Reverses the list

    if (start == NULL) {    // check if the list is empty
        printf("\nList is Empty!");
        return;
    }

    struct node *previousNode, *currentNode, *nextNode;
    previousNode = NULL;
    currentNode = nextNode = start;
    while (nextNode != NULL) {
        nextNode = nextNode->next;
        currentNode->next = previousNode;
        currentNode->previous = nextNode;
        previousNode = currentNode;
        currentNode = nextNode;
    }
    start = previousNode;

}
```

```
void merge() {
   struct node *ptr;
   struct nodeTwo *ptrTwo;
   secondLinkedList();
   ptr = start;
   while (ptr->next != NULL) {
      ptr = ptr->next;
   }
   ptr->next = (struct node *)startTwo;
   startTwo->previousTwo = (struct nodeTwo*)ptr;
   sort();
}

void display() {    // Displays elements of the list

   if (start == NULL) {    // check if list is empty
      printf("\nList is empty!");
      return;
   }
   // traversing pointer
   struct node *ptr = start;

   printf("Elements in the list are  : ");
   while (ptr->next != NULL) {
      printf("%d  ", ptr->data);
      ptr = ptr->next;
   }
   printf("%d", ptr->data);

}

void displayListTwo() {

   struct nodeTwo* ptr;
   ptr = startTwo;
   if (ptr == NULL) {
      printf("\nList is empty!");
      return;
   }
   printf("\n");
   while (ptr->nextTwo != NULL) {
      printf("%d  ", ptr->dataTwo);
      ptr = ptr->nextTwo;
   }
   printf("%d ", ptr->dataTwo);

}


int main() {
```

```
int choice, toInsert, toUpdate, val, pos;

while (1) {

    printf("\n*1   INSERT At END  ");
    printf("\n*2   INSERT At BEGINING  ");
    printf("\n*3   INSERT BEFORE VAL  ");
    printf("\n*4   INSERT AFTER VAL  ");
    printf("\n*5   INSERT At POSITION  ");
    printf("\n*6   DELETE At END  ");
    printf("\n*7   DELETE At BEGINING  ");
    printf("\n*8   DELETE BEFORE VAL  ");
    printf("\n*9   DELETE AFTER VAL  ");
    printf("\n*10  DELETE At POSITION  ");
    printf("\n*11  UPDATE At END  ");
    printf("\n*12  UPDATE At BEGINING  ");
    printf("\n*13  UPDATE BEFORE VAL  ");
    printf("\n*14  UPDATE AFTER VAL  ");
    printf("\n*15  UPDATE At POSITION  ");
    printf("\n*16  SEARCH in the list  ");
    printf("\n*17  COUNT NODE in the list  ");
    printf("\n*18  DISPLAY elements of the list  ");
    printf("\n*19  REVERSE List  ");
    printf("\n*20  SORT List");
    printf("\n*21  MERGE List");
    printf("\n*22  EXIT  ");
    printf("\nEnter your choice :  ");
    scanf("%d", &choice);

    switch (choice) {

        case 1:
            printf("\nEnter element to insert :  ");
            scanf("%d", &toInsert);
            insertAtEnd(toInsert);
            break;

        case 2:
            printf("\nEnter element to insert :  ");
            scanf("%d", &toInsert);
            insertAtBeginning(toInsert);
            break;

        case 3:
            printf("\nEnter element to insert :  ");
            scanf("%d", &toInsert);
            printf("\nEnter value BEFORE which to insert :  ");
            scanf("%d", &val);
            insertBeforeVal(toInsert, val);
            break;

        case 4:
```

```
    printf("\nEnter element to insert :  ");
    scanf("%d", &toInsert);
    printf("\nEnter value AFTER which to insert :  ");
    scanf("%d", &val);
    insertAfterVal(toInsert, val);
    break;

case 5:
    printf("\nEnter element to insert :  ");
    scanf("%d", &toInsert);
    printf("\nEnter POSITION AT which to insert :  ");
    scanf("%d", &pos);
    insertAtPosition(toInsert, pos);
    break;

case 6:
    deleteAtEnd();
    break;

case 7:
    deleteAtBeginning();
    break;

case 8:
    printf("\nEnter value BEFORE which to DELETE :  ");
    scanf("%d", &val);
    deleteBeforeVal(val);
    break;

case 9:
    printf("\nEnter value AFTER which to DELETE :  ");
    scanf("%d", &val);
    deleteAfterVal(val);
    break;

case 10:
    printf("\nEnter POSITION AT which to DELETE :  ");
    scanf("%d", &pos);
    deleteAtPosition(pos);
    break;

case 11:
    printf("\nEnter element to UPDATE :  ");
    scanf("%d", &toUpdate);
    updateAtEnd(toUpdate);
    break;

case 12:
    printf("\nEnter element to UPDATE :  ");
    scanf("%d", &toUpdate);
    updateAtBeginning(toUpdate);
    break;
```

```
case 13:
   printf("\nEnter element to UPDATE :  ");
   scanf("%d", &toUpdate);
   printf("\nEnter value BEFORE which to UPDATE :  ");
   scanf("%d", &val);
   updateBeforeVal(toUpdate, val);
   break;

case 14:
   printf("\nEnter element to UPDATE :  ");
   scanf("%d", &toUpdate);
   printf("\nEnter value AFTER which to UPDATE :  ");
   scanf("%d", &val);
   updateBeforeVal(toUpdate, val);
   break;

case 15:
   printf("\nEnter element to UPDATE :  ");
   scanf("%d", &toUpdate);
   printf("\nEnter POSITION AT which to UPDATE :  ");
   scanf("%d", &pos);
   updateAtPosition(toUpdate, pos);
   break;

case 16:
   printf("\nEnter a value to SEARCH :  ");
   scanf("%d", &val);
   search(val);
   break;

case 17:
   printf("\nList contains  %d  elements", countNodes());
   break;

case 18:
   printf("\nElements in the list are :  ");
   display();
   break;

case 19:
   reverse();
   printf("\nList is reversed");
   break;

case 20:
   sort();
   printf("\nList is sorted");
   break;

case 21:
   merge();
```

```
            printf("\nTwo lists are merged!");
            break;

        case 22:
            printf("*** E X I T I N G ***");
            exit(1);
            break;

        default:
            printf("INVALID INPUT");
      }

   }

   return 0;
}
```

// output

```
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  REVERSE List
*20  SORT List
*21  MERGE List
*22  EXIT
Enter your choice :  1

Enter element to insert :  5

*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  REVERSE List
*20  SORT List
*21  MERGE List
*22  EXIT
```

```
Enter your choice :  1

Enter element to insert :  10

*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  REVERSE List
*20  SORT List
*21  MERGE List
*22  EXIT
Enter your choice :  1

Enter element to insert :  15

*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  REVERSE List
```

```
Enter your choice :  18
Elements in the list are  : 5  10  15
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  REVERSE List
*20  SORT List
*21  MERGE List
*22  EXIT
Enter your choice :  11

Enter element to UPDATE :  200

*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  REVERSE List
*20  SORT List
```

```
Enter your choice :  18
Elements in the list are  : 5  10  200
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  REVERSE List
*20  SORT List
*21  MERGE List
*22  EXIT
Enter your choice :  9

Enter value AFTER which to DELETE :  10

Deleted element is :  200
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  REVERSE List
*20  SORT List
```

```
Enter your choice :  18
Elements in the list are  : 5  10
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  REVERSE List
*20  SORT List
*21  MERGE List
*22  EXIT
Enter your choice :  22
*** E X I T I N G ***
Process returned 1 (0x1)   execution time : 48.254 s
Press any key to continue.
```

**DSA LAB**

**Lab Assignment number 12**

**Name:** Aamir Ansari **Batch:** A **Roll no:** 01

**Aim:** To implement Circular Doubly linked lists

## THEORY:

**Circular Doubly Linked List:**

Circular Doubly Linked List is a linked list in which two consecutive elements are linked or connected by previous and next pointer and the last node points to first node by next pointer and also the first node points to last node by previous pointer.

## ALGORITHM:

1. INSERT

a)At a position:

```
Step 1: [INITIALIZE] temp
Step 2: IF POSITION == 1
            [INITIALIZE] ptr
            SET ptr->data = data
            IF START == NULL
                  SET START = ptr
                  SET START->next =NULL
                  SET START->prev = NULL
                  SET START->next = START->prev = ptr
                  Goto Step 10
            SET START->prev->next = ptr
            SET ptr->prev  = START->prev
            SET START->prev = ptr
            SET ptr->next = START
            SET START=ptr
            Goto Step 10
Step 3: SET temp = START
Step 4: SET i = 0
Step 5:Repaet step 6&7 while i<position-1 AND temp->next != START
Step 6:     SET temp = temp->next
Step 7:     SET i++
Step 8: IF temp == NULL
            PRINT “Less elements”
            Goto Step 10
Step 9: ELSE
            [INITIALIZE] ptr ,emp
            SET ptr->data= data
            SET emp=START
            Repeat step  while emp->data != val
                  SET emp = emp->next
            SET ptr->next = emp->next
            SET ptr->prev = emp
            SET emp->next->prev = ptr
```

SET emp->next = ptr
Step 10:EXIT

b)Before a given value:
Step 1: [INITIALIZE] ptr ,temp
Step 2: SET ptr->data= data
Step 3: SET temp=START
Step 4: IF START == NULL
PRINT “LIST EMPTY”
Goto Step 11
Step 5: Repeat step 6 while temp->data != val
Step 6: SET temp = temp->next
Step 7: SET ptr->next = temp
Step 8: SET ptr->prev = temp->prev
Step 9: SET temp->prev->next = ptr
Step 10: SET temp->prev = ptr
Step 11:EXIT

c)After a given Value:
Step 1: [INITIALIZE] ptr ,temp
Step 2: SET ptr->data= data
Step 3: SET temp=START
Step 4: IF START == NULL
PRINT “LIST EMPTY”
Goto Step 11
Step 5: Repeat step 6 while temp->data != val
Step 6: SET temp = temp->next
Step 7: SET ptr->next = temp->next
Step 8: SET ptr->prev = temp
Step 9: SET temp->next->prev = ptr
Step 10: SET temp->next = ptr
Step 11:EXIT

d)At the beginning
Step 1: [INITIALIZE] ptr
Step 2: SET ptr->data = data
Step 3: IF START == NULL
SET START = ptr
SET START->next =NULL
SET START->prev = NULL
SET START->next = START->prev = ptr
Goto Step 9
Step 4: SET START->prev->next = ptr
Step 5: SET ptr->prev = START->prev
Step 6: SET START->prev = ptr
Step 7: SET ptr->next = START
Step 8: SET START=ptr
Step 9: EXIT

e)At the end

Step 1: [INITIALIZE] ptr,temp
Step 2: SET ptr->data = data
Step 3: IF START == NULL
SET START = ptr
SET START->next = START->prev = ptr
Goto Step 5
Step 4: ELSE
SET temp = START
Repeat step while temp->next != START
temp = temp->next
SET temp->next = ptr
SET ptr ->prev=temp
SET START -> prev = ptr
SET ptr -> next = START
Step 5: EXIT

2.DELETE
a)Value at a particular Position
Step 1: IF START == NULL
PRINT "Linked list is already empty"
Goto Step 12
Step 2: [INITIALIZE] *temp
Step 3: IF position == 1
[INITIALIZE] ptr
SET ptr=START
SET START = START->next
SET START->prev = NULL
free(ptr)
Step 4: SET temp= START
Step 5: SET i=0
Step 6:Repeat step 7 &8 while i<position-1 && temp!=NULL
Step 7: SET temp = temp->next
Step 8: SET i++
Step 9: IF temp == NULL
PRINT “Less nodes”
Goto step 12
Step 11:ELSE
[INITIALIZE] *ptr , *empty
SET ptr= START
Repeat step while ptr != temp->data
SET ptr = ptr->next
SET empty = ptr->next
SET ptr->next = empty->next
SET empty->next->prev = ptr
free(empty)
Step 12:EXIT

b)Before a particular value

```
Step 1: IF START == NULL
                PRINT "Linked list is already empty"
                Goto Step 10
Step 2: [INITIALIZE] *ptr , *temp
Step 3: SET ptr= START
Step 4:Repeat step 5 while ptr->data != val
Step 5:         SET ptr = ptr->next
Step 6: SET temp = ptr->prev
Step 7: SET ptr->prev = temp->prev
Step 8: SET temp->prev->next = ptr
Step 9: free(temp)
Step 10:EXIT
```

c)After a particular value

```
Step 1: IF START == NULL
                PRINT "Linked list is already empty"
                Goto Step 10
Step 2: [INITIALIZE] *ptr , *temp
Step 3: SET ptr= START
Step 4:Repeat step 5 while ptr != val
Step 5:         SET ptr = ptr->next
Step 6: SET temp = ptr->next
Step 7: SET ptr->next = temp->next
Step 8: SET temp->next->prev = ptr
Step 9: free(temp)
Step 10:EXIT
```

d)Value at the beginning

```
Step 1: IF START == NULL
                PRINT "List is empty"
                Goto Step 8
Step 2: [INITIALIZE] ptr
Step 3: SET ptr=START
Step 4: SET ptr->prev->next = ptr->next
Step 5: SET ptr->next->prev = ptr->prev
Step 6: SET START=START->next
Step 7: free(ptr)
Step 8: EXIT
```

e)At the end

```
Step 1: IF START == NULL
                PRINT "List is  empty"
                Goto Step 4
Step 2: ELSE IF START->next == START
                SET START = NULL
                free(START)
Step 3: ELSE
                [INITIALIZE] ptr = START
                Repeat while ptr -> next != NULL
                        SET ptr = ptr->next
```

SET ptr->prev->next = START
SET START->prev = ptr->prev
free(ptr)
Step 4:EXIT

3.UPDATE

a)Value at a given Position

Step 1: IF START == NULL
PRINT "List is empty"
Goto Step 7
Step 3: IF position == 1
SET START->data = data
Step 4: SET temp= START
Step 5: SET i=0
Step 6:Repeat step 7 &8 while i<position-1 && temp!=NULL
Step 7: SET temp = temp->next
Step 8: SET i++
Step 9: IF temp == NULL
PRINT “Less nodes”
Goto step 12
Step 11:ELSE
[INITIALIZE] *ptr=START
Repeat step while ptr->data != temp->data
SET ptr = ptr->next
SET ptr->next->data = data
Step 12:EXIT

b)Before a particular value

Step 1: IF START == NULL
PRINT "List is empty"
Goto Step 6
Step 2: [INITIALIZE] *ptr=START
Step 3:Repeat step 4 while ptr->data != val
Step 4: ptr = ptr->next
Step 5: SET ptr->prev->data = data
Step 6:EXIT

c)After a particular value

Step 1: IF START == NULL
PRINT "Linked list is already empty"
Goto Step 6
Step 2: [INITIALIZE] *ptr=START
Step 3:Repeat step 4 while ptr->data != val
Step 4: SET ptr = ptr->next
Step 5:SET ptr->next->data = data
Step 6: EXIT

d)Value at the beginning

Step 1: IF START == NULL
PRINT "List is empty"
Goto Step 3
Step 2: SET START->data = data
Step 3:EXIT

e)At the end
Step 1: IF START == NULL
PRINT "Linked list is already empty"
Goto Step 4
Step 2: [INITIALIZE] *ptr=START->prev
Step 3:SET ptr->data = data
Step 4:EXIT

4.SEARCH
Step 1: IF START == NULL
PRINT "List is empty"
Goto Step 9
Step 2: [INITIALIZE] *ptr = START
Step 3: SET count =1
Step 4: Repeat Step 5&6 while ptr->data != data && count<=countNodes()+1
Step 5: SET ptr = ptr->next
Step 6: SET count=1
Step 7: IF count>countNodes()
PRINT 'ELEMENT NOT FOUND'
Step 8: ELSE
PRINT "Element found at the position"
Step 9: EXIT

5.COUNT NODES
Step 1:[ INITIALIZE] *ptr = START
Step 2: SET count = 1
Step 3:Repeat step 4&5 while ptr->next !=NULL
Step 4: SET count++
Step 5: SET ptr=ptr->next
Step 6 :RETURN count
Step 7:EXIT

7.DISPLAY
Step 1:[INITIALIZE] *ptr =START
Step 2:Repeat step 3&4 while ptr != NULL
Step 3: PRINT (ptr->data)
Step 4: SET ptr = ptr->next
Step 5: PRINT (ptr->data)
Step 6: EXIT

**DSA LAB**
**Lab Assignment number 12**

**Name:** Aamir Ansari **Batch:** A **Roll no:** 01

**Aim:** Implementation of Circular Doubly linked list

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

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

struct node *start = NULL;

int countNodes() {

   if (start == NULL) {
      printf("\nLIST IS EMPTY!");
      return 0;
   }
   struct node *ptr = start;
   int count = 1;

   while (ptr->next != start) {
      ptr = ptr->next;
      count++;
   }
   return count;
}

void insertAtBeginning(int toInsert) {    // Inserts node at the beginning

   struct node *newNode;
   newNode = (struct node *)malloc(sizeof(struct node));
   newNode->data = toInsert;

   if (start == NULL) {    // first node of the list is added
      newNode->next = newNode;
      newNode->previous = newNode;
      start = newNode;
   } else {
      // linking last node with new node
      newNode->previous = start->previous;
      start->previous->next = newNode;
```

```
        // linking new node with current first node
        newNode->next = start;
        start->previous = newNode;
        start = newNode;
    }
}

void insertAtEnd(int toInsert) {    // Inserts node at the end of the list

    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = toInsert;

    if (start == NULL) {    // first node of the list is added
        newNode->next = newNode;
        newNode->previous = newNode;
        start = newNode;
    } else {
        // linking newNode with current last node
        newNode->previous = start->previous;
        start->previous->next = newNode;

        // linking newNode with start node
        newNode->next = start;
        start->previous = newNode;
    }
}

void insertBeforeVal(int toInsert, int val) {    // Inserts before val

    if (start == NULL) {
        printf("\nLIST IS EMPTY!");
        return;
    }
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = toInsert;
    struct node *ptr = start;

    if (ptr->data == val) {    // inserting before current first node
        insertAtBeginning(toInsert);
    } else {
        while (ptr->next->data != val) {    // traversing
            ptr = ptr->next;
        }
        // linking newNode with ptr->next
        newNode->next = ptr->next;
        ptr->next->previous = newNode;
        // linking newNode with ptr
        newNode->previous = ptr;
        ptr->next = newNode;
    }
```

```
}

void insertAfterVal(int toInsert, int val) {    // Inserts after val

    if (start == NULL) {
        printf("\nLIST IS EMPTY!");
        return;
    }
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = toInsert;
    struct node *ptr = start;

    while (ptr->data != val) {
        ptr = ptr->next;
    }

    if (ptr == start->previous) {    // inserting after current last node
        insertAtEnd(toInsert);
    } else {
        // linking newNode with ptr->next
        newNode->next = ptr->next;
        ptr->next->previous = newNode;
        // linking newNode with ptr
        newNode->previous = ptr;
        ptr->next = newNode;
    }
}

void insertAtPosition(int toInsert, int pos) {

    if (start == NULL) {
        printf("\nLIST IS EMPTY!");
        return;
    }
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = toInsert;
    struct node *ptr = start;
    int count = 1;

    while (count != pos && ptr->next != start) {    // traversing
        ptr = ptr->next;
        count++;
    }
    if (pos > count+1) {    // invalid position
        printf("\nList is not that long!");
        return;
    }

    if (count == 1) {    // adding new node before first node
        insertAtBeginning(toInsert);
```

```
    } else if (ptr->next == start && count < pos) {    // inserting after last node  /* second condition =>
when the postion is second-last */
        insertAtEnd(toInsert);
    } else {    // inserting at any position
        insertBeforeVal(toInsert, ptr->data);
    }
}

void deleteAtBeginning() {    // Deletes elements at the beginning

    if (start == NULL) {
        printf("\nLIST IS EMPTY!");
        return;
    }
    struct node *ptr = start;

    printf("\nDeleted element is :  %d", ptr->data);
    if (start->next == start) {    // deleting only remaining node
        free(start);
        start = NULL;
    } else {
        // linking current second node with the last node
        start->next->previous = start->previous;
        start->previous->next = start->next;
        // shifting start
        start = start->next;
        // freeing first node
        free(ptr);
    }
}

void deleteAtEnd() {    // deletes element at the end

    if (start == NULL) {
        printf("\nLIST IS EMPTY!");
        return;
    }
    struct node *ptr = start;

    printf("\nDeleted element is :  %d", ptr->previous->data);
    if (start->next == start) {    // deleting only remaining node
        free(start);
        start = NULL;
    } else {
        // shifting ptr to last node
        ptr = ptr->previous;
        // linking current second last node to start
        ptr->previous->next = start;
        start->previous = ptr->previous;
        // freeing last node
        free(ptr);
    }
```

```
}

void deleteBeforeVal(int val) {    // Deletes before val

    if (start == NULL) {
        printf("\nLIST IS EMPTY!");
        return;
    }
    struct node *ptr = start;

    if (ptr->data == val) {    // deleting before first node
        deleteAtEnd();
    } else {
        while (ptr->next->data != val) {    // traversing
            ptr = ptr->next;
        }
        printf("\nDeleted element is :  %d", ptr->data);
        // linking nodes which are before and after ptr
        ptr->previous->next = ptr->next;
        ptr->next->previous = ptr->previous;
        // freeing ptr
        free(ptr);
    }
}

void deleteAfterVal(int val) {    // deletes after val

    if (start == NULL) {
        printf("\nLIST IS EMPTY!");
        return;
    }
    struct node *ptr = start;

    while (ptr->data != val) {
        ptr = ptr->next;
    }

    if (ptr->next == start) {    // deleting after last node
        deleteAtBeginning();
    } else if (ptr->next->next == start) {    // deleting last node
        deleteAtEnd();
    } else {
        // shifting ptr to node which is to be deleted
        ptr = ptr->next;
        deleteBeforeVal(ptr->next->data);
    }
}

void deleteAtPosition(int pos) {    // deletes at position

    if (start == NULL) {
        printf("\nLIST IS EMPTY!");
```

```
        return;
    }
    struct node *ptr = start;
    int count = 1;

    while (count!=pos && ptr->next!=start) {
        ptr = ptr->next;
        count++;
    }
    if (pos > count) {
        printf("\nINVALID POSITION!");
        return;
    }

    if (count == 1) {    // deleting first node
        deleteAtBeginning();
    } else if (ptr->next == start) {    // deleting last node
        deleteAtEnd();
    } else {
        deleteAfterVal(ptr->previous->data);
    }

    if (ptr->next == start && ptr->previous == start) {
        start = NULL;
    }
}

void updateAtBeginning(int toUpdate) {    // updates at the beginning

    if (start == NULL) {
        printf("\nLIST IS EMPTY!");
        return;
    }
    // updation
    start->data = toUpdate;
}

void updateAtEnd(int toUpdate) {    // update at the end

    if (start == NULL) {
        printf("\nLIST IS EMPTY!");
        return;
    }
    // updation
    start->previous->data = toUpdate;
}

void updateBeforeVal(int toUpdate, int val) {    // updates before val

    if (start == NULL) {
        printf("\nLIST IS EMPTY!");
        return;
```

```
    }
    struct node *ptr = start;

    if (ptr->data == val) {    // updating before first node
        start->previous->data = toUpdate;
    } else {
        while (ptr->next->data != val) {    // traversing
            ptr = ptr->next;
        }
        // updation
        ptr->data = toUpdate;
    }
}

void updateAfterVal(int toUpdate, int val) {    // update after val is encountered

    if (start == NULL) {
        printf("\nLIST IS EMPTY!");
        return;
    }
    struct node *ptr = start;

    while (ptr->data != val) {
        ptr = ptr->next;
    }
    ptr = ptr->next;
    ptr->data = toUpdate;
}

void updateAtPosition(int toUpdate, int pos) {    // updates at posiotion

    if (start == NULL) {
        printf("\nLIST IS EMPTY!");
        return;
    }
    struct node *ptr = start;
    int count = 1;

    while (count!=pos && ptr->next != start) {
        ptr = ptr->next;
        count++;
    }

    if (pos>count) {
        printf("\nINVALID POSITION!");
        return;
    }

    ptr->data = toUpdate;
}
```

```
void search(int val) {

    if (start == NULL) {
        printf("\nLIST IS EMPTY!");
        return;
    }
    struct node *ptr = start;
    int count = 1;

    while (ptr->data != val && count<=countNodes()+1) {
        ptr = ptr->next;
        count++;
    }

    // printing
    if (count > countNodes()) {
        printf("\n%d is not present in the list!", val);
    } else {
        printf("\nPosition of %d in the list is :  %d", val, count);
    }
}

void display() {

    if (start == NULL) {    // check if list is empty
        printf("\nList is empty!");
        return;
    }
    // traversing pointer
    struct node *ptr = start;

    printf("FORWARD :  ");
    while (ptr->next != start) {
        printf("%d  ", ptr->data);
        ptr = ptr->next;
    }
    printf("%d", ptr->data);

    printf("\nREVERSE :  ");
    while (ptr->previous != start->previous) {
        printf("%d  ", ptr->data);
        ptr = ptr->previous;
    }
    printf("%d", ptr->data);
}

int main() {

    int choice, toInsert, toUpdate, val, pos;

    while (1) {
```

```
printf("\n*1   INSERT At END  ");
printf("\n*2   INSERT At BEGINING  ");
printf("\n*3   INSERT BEFORE VAL  ");
printf("\n*4   INSERT AFTER VAL  ");
printf("\n*5   INSERT At POSITION  ");
printf("\n*6   DELETE At END  ");
printf("\n*7   DELETE At BEGINING  ");
printf("\n*8   DELETE BEFORE VAL  ");
printf("\n*9   DELETE AFTER VAL  ");
printf("\n*10  DELETE At POSITION  ");
printf("\n*11  UPDATE At END  ");
printf("\n*12  UPDATE At BEGINING  ");
printf("\n*13  UPDATE BEFORE VAL  ");
printf("\n*14  UPDATE AFTER VAL  ");
printf("\n*15  UPDATE At POSITION  ");
printf("\n*16  SEARCH in the list  ");
printf("\n*17  COUNT NODE in the list  ");
printf("\n*18  DISPLAY elements of the list  ");
printf("\n*19  EXIT  ");
printf("\nEnter your choice :  ");
scanf("%d", &choice);

switch (choice) {

   case 1:
      printf("\nEnter element to insert :  ");
      scanf("%d", &toInsert);
      insertAtEnd(toInsert);
      break;

   case 2:
      printf("\nEnter element to insert :  ");
      scanf("%d", &toInsert);
      insertAtBeginning(toInsert);
      break;

   case 3:
      printf("\nEnter element to insert :  ");
      scanf("%d", &toInsert);
      printf("\nEnter value BEFORE which to insert :  ");
      scanf("%d", &val);
      insertBeforeVal(toInsert, val);
      break;

   case 4:
      printf("\nEnter element to insert :  ");
      scanf("%d", &toInsert);
      printf("\nEnter value AFTER which to insert :  ");
      scanf("%d", &val);
      insertAfterVal(toInsert, val);
      break;
```

```
case 5:
    printf("\nEnter element to insert :  ");
    scanf("%d", &toInsert);
    printf("\nEnter POSITION AT which to insert :  ");
    scanf("%d", &pos);
    insertAtPosition(toInsert, pos);
    break;

case 6:
    deleteAtEnd();
    break;

case 7:
    deleteAtBeginning();
    break;

case 8:
    printf("\nEnter value BEFORE which to DELETE :  ");
    scanf("%d", &val);
    deleteBeforeVal(val);
    break;

case 9:
    printf("\nEnter value AFTER which to DELETE :  ");
    scanf("%d", &val);
    deleteAfterVal(val);
    break;

case 10:
    printf("\nEnter POSITION AT which to DELETE :  ");
    scanf("%d", &pos);
    deleteAtPosition(pos);
    break;

case 11:
    printf("\nEnter element to UPDATE :  ");
    scanf("%d", &toUpdate);
    updateAtEnd(toUpdate);
    break;

case 12:
    printf("\nEnter element to UPDATE :  ");
    scanf("%d", &toUpdate);
    updateAtBeginning(toUpdate);
    break;

case 13:
    printf("\nEnter element to UPDATE :  ");
    scanf("%d", &toUpdate);
    printf("\nEnter value BEFORE which to UPDATE :  ");
    scanf("%d", &val);
```

```
            updateBeforeVal(toUpdate, val);
            break;

        case 14:
            printf("\nEnter element to UPDATE :  ");
            scanf("%d", &toUpdate);
            printf("\nEnter value AFTER which to UPDATE :  ");
            scanf("%d", &val);
            updateBeforeVal(toUpdate, val);
            break;

        case 15:
            printf("\nEnter element to UPDATE :  ");
            scanf("%d", &toUpdate);
            printf("\nEnter POSITION AT which to UPDATE :  ");
            scanf("%d", &pos);
            updateAtPosition(toUpdate, pos);
            break;

        case 16:
            printf("\nEnter a value to SEARCH :  ");
            scanf("%d", &val);
            search(val);
            break;

        case 17:
            printf("\nList contains  %d  elements", countNodes());
            break;

        case 18:
            printf("\nElements in the list are :  ");
            display();
            break;

        case 19:
            printf("*** E X I T I N G ***");
            exit(1);
            break;

        default:
            printf("INVALID INPUT");
      }
   }
   return 0;
}
```

// output

```
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  EXIT
Enter your choice :  1

Enter element to insert :  5

*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  EXIT
Enter your choice :  1

Enter element to insert :  10
```

```
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  EXIT
Enter your choice :  1

Enter element to insert :  15

*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  EXIT
Enter your choice :  18

Elements in the list are :  FORWARD :  5  10  15
REVERSE :  15  10  5
```

```
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  EXIT
Enter your choice :  16

Enter a value to SEARCH :  10

Position of 10 in the list is :  2
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  EXIT
Enter your choice :  17

List contains  3  elements
```

```
*1   INSERT At END
*2   INSERT At BEGINING
*3   INSERT BEFORE VAL
*4   INSERT AFTER VAL
*5   INSERT At POSITION
*6   DELETE At END
*7   DELETE At BEGINING
*8   DELETE BEFORE VAL
*9   DELETE AFTER VAL
*10  DELETE At POSITION
*11  UPDATE At END
*12  UPDATE At BEGINING
*13  UPDATE BEFORE VAL
*14  UPDATE AFTER VAL
*15  UPDATE At POSITION
*16  SEARCH in the list
*17  COUNT NODE in the list
*18  DISPLAY elements of the list
*19  EXIT
Enter your choice :  19
*** E X I T I N G ***
Process returned 1 (0x1)   execution time : 37.413 s
Press any key to continue.
```

**DSA LAB**
**Lab Assignment number 11**

**Name:** Aamir Ansari **Batch:** A **Roll no:** 01

**Aim:** Implementation of various operations on binary search tree

## Algorithms:

### Create Node:

getNewNode (data)
Step 1: [INITIALIZE] newNode
Step 2: SET newNode -> data = data
Step 3: SET newNode -> left = NULL
Step 4: SET newNode -> right = NULL
Stem 5: return newNode
Step 6: EXIT

### Insertion of node:

```
Insert (ROOT, VAL)
Step 1: IF ROOT = NULL, then
            Allocate memory for ROOT
            SET ROOT->DATA = VAL
            SET ROOT->LEFT = ROOT ->RIGHT = NULL
        ELSE
            IF VAL < ROOT->DATA
                ROOT->LEFT= Insert(ROOT->LEFT, VAL)
            ELSE
                ROOT->RIGHT=Insert(ROOT->RIGHT, VAL)
            [END OF IF]
        [END OF IF]
Step 2: End
```

### Deletion of node:

```
Delete (ROOT, VAL)
Step 1: IF ROOT = NULL, then
            return ROOT
        IF VAL < ROOT->DATA
            ROOT->LEFT=Delete(ROOT->LEFT, VAL)
        ELSE IF VAL > ROOT->DATA
            ROOT->RIGHT=Delete(ROOT->RIGHT, VAL)
        ELSE
```

```
            // if node is leaf node or single child node
            IF ROOT->LEFT = NULL
                  TEMP=ROOT->RIGHT
                  FREE ROOT
                  RETURN TEMP
            ELSE IF ROOT->RIGHT=NULL
                  TEMP=ROOT->LEFT
                  FREE ROOT
                  RETURN TEMP
            ELSE
                  // If node has both left and right child
                  SET TEMP = findLargestNode(ROOT->LEFT) //inorder predecessor
                  SET ROOT->DATA = TEMP->DATA
                  ROOT->LEFT=Delete (ROOT->LEFT, TEMP->DATA)
            [END OF IF]
      [END OF IF]
Step 2:  RETURN ROOT
Step 3:  End
```

## Searching for data:

```
searchElement (ROOT, VAL)
Step 1: IF ROOT ->DATA = VAL OR ROOT = NULL, then
            Return ROOT
      ELSE
            IF VAL < ROOT ->DATA
                  Return searchElement(ROOT->LEFT,VAL)
            ELSE
                  Return searchElement(ROOT->RIGHT,VAL)
            [END OF IF]
      [END OF IF]
Step 2: End
```

## Height:

```
Height (ROOT)
Step 1: IF ROOT = NULL, then
            Return 0
      ELSE
            SET LeftHeight = Height(ROOT ->LEFT)
            SET RightHeight = Height(ROOT ->RIGHT)
```

```
            IF LeftHeight > RightHeight
                  Return LeftHeight + 1
            ELSE
                  Return RightHeight + 1
            [END OF IF]
      [END OF IF]Step 2: End
```

## In-order Traversal:

```
inorderTraversal(root)
STEP 1:  IF ROOT != NULL
            inorderTraversal(root->left);
            printf("%d\t", root->data);
            inorderTraversal(root->right);
Step 2:  EXIT
```

## Pre-order Traversal:

```
preorderTraversal(root)
STEP 1:  IF ROOT != NULL
            printf("%d\t", root->data);
            preorderTraversal(root->left);
            preinorderTraversal(root->right);
Step 2:  EXIT
```

## Post-order Traversal:

```
postorderTraversal(root)
STEP 1:  IF ROOT != NULL
            postorderTraversal(root->left);
            postorderTraversal(root->right);
            printf("%d\t", root->data);
Step 2:  EXIT
```

## Count nodes:

```
totalNodes (ROOT)
Step 1: IF ROOT = NULL, then
            Return 0
      ELSE
            Return totalNodes(ROOT ->LEFT) + totalNodes(ROOT ->RIGHT) + 1
      [END OF IF]
Step 2: End
```

## Count Leaf nodes:

```
countLeafNodes(ROOT)
Step 1:  IF ROOT = NULL THEN
                return 0
         [END IF]
Step 2:  IF ROOT -> left = ROOT -> RIGHT = NULL THEN
                return 1
         ELSE
                return countLeafNodes(ROOT->left) + countLeafNodes(ROOT->right)
         [END IF]
Step 3:  EXIT
```

## Count Non-leaf Nodes:

```
countNonLeafNodes(ROOT)
Step 1:  return countAllNodes(ROOT) – countLeafNodes(ROOT)
Step 2:  EXIT
```

## Find Minimum:

```
findMin(ROOT)
Step 1:  Repeat step 2 while ROOT->LEFT != NULL
Step 2:          SET ROOT = ROOT -> LEFT
Step 3:  return ROOT
Step 4:  EXIT
```

## Find Maximum:

```
findMax(ROOT)
Step 1:  Repeat step 2 while ROOT->RIGHT != NULL
Step 2:          SET ROOT = ROOT -> RIGHT
Step 3:  return ROOT
Step 4:  EXIT
```

## Mirror image:

```
mirrorImage(ROOT)
Step 1:  [INITIALIZE] ptr
Step 2:  IF ROOT != NULL
Step 3:  mirrorImage(root–>left)
Step 4:  mirrorImage(root–>right)
Step 5:  ptr=ROOT–>left
Step 6:  ptr–>left = ptr–>right
Step 7:  ROOT–>right = ptr
Step 8:  EXIT
```

## Deleting complete tree:

```
deleteTree(ROOT)
Step 1: IF ROOT != NULL , THEN
                deleteTree (ROOT ->LEFT)
                deleteTree (ROOT ->RIGHT)
                Free (ROOT)
        [END OF IF]
Step 2: End
```

## Implementation of Binary Search Tree

// code

```
#include <stdio.h>
#include <stdlib.h>
// Declaration of node of tree
struct node {
    struct node *left;
    int data;
    struct node *right;
};

// declaring root node
struct node *root = NULL;

struct node *findMax(struct node *root) {
    while (root->right != NULL) {
        root = root->right;
    }
    return root;
}

struct node *findMin(struct node *root) {
    while (root->left != NULL) {
        root = root->left;
    }
    return root;
}

struct node *getNewNode(int data) {    // initialises and allocates memory for newNode
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data  = data;
    newNode->left  = NULL;
    newNode->right = NULL;
    return newNode;
}

struct node *insert(struct node *root, int data) {
    if (root == NULL) {    // when tree is empty
        root = getNewNode(data);
        return root;
    }
    if (data <= root->data) {    // inserting in left subtree
        root->left = insert(root->left, data);
    }
    else {    // inserting in right subtree
        root->right = insert(root->right, data);
    }
    // returning original root of the tree
    return root;
```

```
}

struct node *delete(struct node *root, int val) {
   if (root == NULL) {    // empty tree
      return root;
   }
   else if (val < root->data) {    // finding node in left sub-tree
      root->left = delete (root->left, val);
   }
   else if (val > root->data) {    // finding node in right sub-tree
      root->right = delete (root->right, val);
   }
   else {    // found the node
      if (root->right == NULL && root->left == NULL) {    // deleting leaf node
         free(root);
         root = NULL;
      } else if (root->right == NULL) {    // deleting a node with only left sub-tree
         struct node *temp = root;
         root = root->left;
         free(temp);
      } else if (root->left == NULL) {    // deleting a node with only right sub-tree
         struct node *temp = root;
         root = root->right;
         free(temp);
      } else {    // deleting nodes with two sub-trees
         // storing address of node with min value in right sub-tree
         struct node *temp = findMin(root->right);
         root->data = temp->data;
         root->right = delete (root->right, temp->data);
      }
   }
   return root;
}

void search(struct node *root, int val) {
   if (root->data == val) {
      printf("\n%d is present in the tree", val);
      return;
   }
   if ((root->right == NULL && root->left == NULL) || root == NULL) {
      printf("\nNot present");
      return;
   }
   if (val <= root->data) {    // search in left sub-tree
      search(root->left, val);
   }
   else {    // search in right sub-tree
      search(root->right, val);
   }
}

int height(struct node *root) {
```

```
    int leftHeight, rightHeight;
    if (root == NULL) {
        return 0;
    }
    else {
        leftHeight = height(root->left);
        rightHeight = height(root->right);

        return (leftHeight > rightHeight) ? leftHeight + 1 : rightHeight + 1;
    }
}

int countAllNodes(struct node *root) {
    if (root == NULL) {
        return 0;
    }
    else {
        return countAllNodes(root->left) + countAllNodes(root->right) + 1;
    }
}

int countLeafNodes(struct node *root) {
    if (root == NULL) {
        return 0;
    }
    else if (root->left == NULL && root->right == NULL) {
        return 1;
    }
    else {
        return countLeafNodes(root->left) + countLeafNodes(root->right);
    }
}

int countNonLeafNodes(struct node *root) {
    return (countAllNodes(root) - countLeafNodes(root));
}

void printOneLevel(struct node *root, int level) {    // print elements on given level
    if (root == NULL) {
        return;
    }
    if (level == 1) {
        printf("%d ", root->data);
    }
    else if (level > 1)  {
        printOneLevel(root->left, level-1);
        printOneLevel(root->right, level-1);
    }
}
void printCompleteTree(struct node *root)  {    // calls printOneLevel for all the levels in the trr
    int h = height(root);
    int i;
```

```
    for (i=1 ; i<=h ; i++) {
        printOneLevel(root, i);
        printf("\n");
    }
}

void mirrorTree(struct node *root) {
    if (root == NULL) {
        return;
    }
    struct node *temp = root;
    // get to all nodes of tree
    mirrorTree(root->left);
    mirrorTree(root->right);
    // swap the pointer
    temp = root->left;
    root->left = root->right;
    root->right = temp;
}

struct node *deleteCompleteTree(struct node *root) {
    if (root != NULL) {
        deleteCompleteTree(root->left);
        deleteCompleteTree(root->right);
        free(root);
    }
}

void preOrderTraversal(struct node *root) {
    if (root == NULL) {
        return;
    }
    // print the data of the node
    printf("%d  ", root->data);

    // recursion on left sub-tree
    preOrderTraversal(root->left);

    //recursion on right sub-tree
    preOrderTraversal(root->right);
}

void inOrderTraversal(struct node *root) {
    if (root == NULL) {
        return;
    }
    // recursion on left sub-tree
    inOrderTraversal(root->left);

    // print the data of the node
    printf("%d  ", root->data);
```

```
    //recursion on right sub-tree
    inOrderTraversal(root->right);

}

void postOrderTraversal(struct node *root) {
    if (root == NULL) {
        return;
    }
    // recursion on left sub-tree
    postOrderTraversal(root->left);

    //recursion on right sub-tree
    postOrderTraversal(root->right);

    // print the data of the node
    printf("%d  ", root->data);
}



int main() {

    struct node *temp;
    int data, i, choice, val;

    while (1) {
        printf("\n(1)  Insert");
        printf("\n(2)  Delete");
        printf("\n(3)  Search");
        printf("\n(4)  Height");
        printf("\n(5)  INORDER");
        printf("\n(6)  PREORDER");
        printf("\n(7)  POSTORDER");
        printf("\n(8)  TOTAL number of nodes");
        printf("\n(9)  Number of LEAF nodes");
        printf("\n(10) Number of NON-LEAF nodes");
        printf("\n(11) Find MIN");
        printf("\n(12) Find MAX");
        printf("\n(13) Display");
        printf("\n(14) Mirror");
        printf("\n(15) Excise Tree");
        printf("\n(16) EXIT");
        printf("\nEnter your choice :  ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("\nEnter data to insert :  ");
                scanf("%d", &data);
                root = insert(root, data);
                printf("\n%d is inserted!", data);
```

```
    break;

case 2:
    printf("\nEnter a value to delete :  ");
    scanf("%d", &val);
    root = delete (root, val);
    printf("\n%d is deleted!", val);
    break;

case 3:
    printf("\nEnter a number to Search");
    scanf("%d", &data);
    search(root, data);
    break;

case 4:
    printf("\nHeight of tree is :  %d", height(root));
    break;

case 5:
    printf("\nIN-ORDER :  ");
    inOrderTraversal(root);
    break;

case 6:
    printf("\nPRE-ORDER :  ");
    preOrderTraversal(root);
    break;

case 7:
    printf("\nPOST-ORDER :  ");
    postOrderTraversal(root);
    break;

case 8:
    printf("\nTotal number of nodes :  %d", countAllNodes(root));
    break;

case 9:
    printf("\nNumber of LEAF nodes :  %d", countLeafNodes(root));
    break;

case 10:
    printf("\nNumber of NON-LEAF nodes :  %d", countNonLeafNodes(root));
    break;

case 11:
    temp = findMin(root);
    printf("\nMINIMUM in tree :  %d", temp->data);
    break;

case 12:
```

```
            temp = findMax(root);
            printf("\nMAXIMUM in tree :  %d", temp->data);
            break;

        case 13:
            printf("\n***TREE***\n");
            printCompleteTree(root);
            break;

        case 14:
            printf("\n***MIRROR***\n");
            mirrorTree(root);
            printCompleteTree(root);
            break;

        case 15:
            deleteCompleteTree(root);
            printf("\nEntire tree is deleted! you happy now, huh?");
            break;
        case 16:
            printf("\n*** E X I T I N G ***\n");
            exit(1);
            break;

        default:
            printf("\n*** I N V A L I D ***");
      }
   }
   return 0;
}
```

// output

```
(1)  Insert
(2)  Delete
(3)  Search
(4)  Height
(5)  INORDER
(6)  PREORDER
(7)  POSTORDER
(8)  TOTAL number of nodes
(9)  Number of LEAF nodes
(10) Number of NON-LEAF nodes
(11) Find MIN
(12) Find MAX
(13) Display
(14) Mirror
(15) Excise Tree
(16) EXIT
Enter your choice :  1

Enter data to insert :  10

10 is inserted!
(1)  Insert
(2)  Delete
(3)  Search
(4)  Height
(5)  INORDER
(6)  PREORDER
(7)  POSTORDER
(8)  TOTAL number of nodes
(9)  Number of LEAF nodes
(10) Number of NON-LEAF nodes
(11) Find MIN
(12) Find MAX
(13) Display
(14) Mirror
(15) Excise Tree
(16) EXIT
Enter your choice :  1

Enter data to insert :  5
```

```
5 is inserted!
(1)  Insert
(2)  Delete
(3)  Search
(4)  Height
(5)  INORDER
(6)  PREORDER
(7)  POSTORDER
(8)  TOTAL number of nodes
(9)  Number of LEAF nodes
(10) Number of NON-LEAF nodes
(11) Find MIN
(12) Find MAX
(13) Display
(14) Mirror
(15) Excise Tree
(16) EXIT
Enter your choice :  1

Enter data to insert :  15

15 is inserted!
(1)  Insert
(2)  Delete
(3)  Search
(4)  Height
(5)  INORDER
(6)  PREORDER
(7)  POSTORDER
(8)  TOTAL number of nodes
(9)  Number of LEAF nodes
(10) Number of NON-LEAF nodes
(11) Find MIN
(12) Find MAX
(13) Display
(14) Mirror
(15) Excise Tree
(16) EXIT
Enter your choice :  1

Enter data to insert :  3
```

```
(1)  Insert
(2)  Delete
(3)  Search
(4)  Height
(5)  INORDER
(6)  PREORDER
(7)  POSTORDER
(8)  TOTAL number of nodes
(9)  Number of LEAF nodes
(10) Number of NON-LEAF nodes
(11) Find MIN
(12) Find MAX
(13) Display
(14) Mirror
(15) Excise Tree
(16) EXIT
Enter your choice :  1

Enter data to insert :  17

17 is inserted!
(1)  Insert
(2)  Delete
(3)  Search
(4)  Height
(5)  INORDER
(6)  PREORDER
(7)  POSTORDER
(8)  TOTAL number of nodes
(9)  Number of LEAF nodes
(10) Number of NON-LEAF nodes
(11) Find MIN
(12) Find MAX
(13) Display
(14) Mirror
(15) Excise Tree
(16) EXIT
Enter your choice :  13
```

```
***TREE***
10
5 15
3 17

(1)  Insert
(2)  Delete
(3)  Search
(4)  Height
(5)  INORDER
(6)  PREORDER
(7)  POSTORDER
(8)  TOTAL number of nodes
(9)  Number of LEAF nodes
(10) Number of NON-LEAF nodes
(11) Find MIN
(12) Find MAX
(13) Display
(14) Mirror
(15) Excise Tree
(16) EXIT
Enter your choice :  2

Enter a value to delete :  5
```

```
5 is deleted!
(1)  Insert
(2)  Delete
(3)  Search
(4)  Height
(5)  INORDER
(6)  PREORDER
(7)  POSTORDER
(8)  TOTAL number of nodes
(9)  Number of LEAF nodes
(10) Number of NON-LEAF nodes
(11) Find MIN
(12) Find MAX
(13) Display
(14) Mirror
(15) Excise Tree
(16) EXIT
Enter your choice :  13

***TREE***
10
3 15
17

(1)  Insert
(2)  Delete
(3)  Search
(4)  Height
(5)  INORDER
(6)  PREORDER
(7)  POSTORDER
(8)  TOTAL number of nodes
(9)  Number of LEAF nodes
(10) Number of NON-LEAF nodes
(11) Find MIN
(12) Find MAX
(13) Display
(14) Mirror
(15) Excise Tree
(16) EXIT
```

```
(1)  Insert
(2)  Delete
(3)  Search
(4)  Height
(5)  INORDER
(6)  PREORDER
(7)  POSTORDER
(8)  TOTAL number of nodes
(9)  Number of LEAF nodes
(10) Number of NON-LEAF nodes
(11) Find MIN
(12) Find MAX
(13) Display
(14) Mirror
(15) Excise Tree
(16) EXIT
Enter your choice :  5

IN-ORDER :  3  10  15  17
(1)  Insert
(2)  Delete
(3)  Search
(4)  Height
(5)  INORDER
(6)  PREORDER
(7)  POSTORDER
(8)  TOTAL number of nodes
(9)  Number of LEAF nodes
(10) Number of NON-LEAF nodes
(11) Find MIN
(12) Find MAX
(13) Display
(14) Mirror
(15) Excise Tree
(16) EXIT
Enter your choice :  6

PRE-ORDER :  10  3  15  17
```

```
POST-ORDER :  3  17  15  10
(1)  Insert
(2)  Delete
(3)  Search
(4)  Height
(5)  INORDER
(6)  PREORDER
(7)  POSTORDER
(8)  TOTAL number of nodes
(9)  Number of LEAF nodes
(10) Number of NON-LEAF nodes
(11) Find MIN
(12) Find MAX
(13) Display
(14) Mirror
(15) Excise Tree
(16) EXIT
Enter your choice :  8

Total number of nodes :  4
(1)  Insert
(2)  Delete
(3)  Search
(4)  Height
(5)  INORDER
(6)  PREORDER
(7)  POSTORDER
(8)  TOTAL number of nodes
(9)  Number of LEAF nodes
(10) Number of NON-LEAF nodes
(11) Find MIN
(12) Find MAX
(13) Display
(14) Mirror
(15) Excise Tree
(16) EXIT
Enter your choice :  14
```

```
***MIRROR***
10
15 3
17

(1)  Insert
(2)  Delete
(3)  Search
(4)  Height
(5)  INORDER
(6)  PREORDER
(7)  POSTORDER
(8)  TOTAL number of nodes
(9)  Number of LEAF nodes
(10) Number of NON-LEAF nodes
(11) Find MIN
(12) Find MAX
(13) Display
(14) Mirror
(15) Excise Tree
(16) EXIT
Enter your choice :  15

Entire tree is deleted! you happy now, huh?

(1)  Insert
(2)  Delete
(3)  Search
(4)  Height
(5)  INORDER
(6)  PREORDER
(7)  POSTORDER
(8)  TOTAL number of nodes
(9)  Number of LEAF nodes
(10) Number of NON-LEAF nodes
(11) Find MIN
(12) Find MAX
(13) Display
(14) Mirror
(15) Excise Tree
(16) EXIT
Enter your choice :  16

*** E X I T I N G ***
```

**DSA LAB**
**Lab Assignment number 14**

**Name:** Aamir Ansari **Batch:** A **Roll no:** 01

**Aim:** To implement various operations on AVL tree

## Theory:

## Algorithm to insert a node in AVL tree:

insertNode (ROOT, DATA)

Step 1: IF ROOT = NULL, then
Allocate memory for newNode
newNode->DATA=DATA
newNode->LEFT=newNode->RIGHT=NULL
newNode->HEIGHT=1
RETURN newNode;
[END OF IF]
Step 2: IF DATA < ROOT->DATA
ROOT->LEFT=insertNode(ROOT->LEFT, DATA)
ELSE IF DATA > ROOT->DATA
ROOT->RIGHT=insertNode(ROOT->RIGHT,DATA)
ELSE
RETURN ROOT
[END OF IF]
Step 3: ROOT->HEIGHT = 1 + MAX ((height(ROOT->LEFT),height(ROOT->RIGHT))
Step 4: SET BALANCE = ROOT->LEFT->HEIGHT – ROOT->RIGHT->HEIGHT
Step 5: IF BALANCE>1 AND DATALEFT->DATA
RETURN rotateRight(ROOT)
IF BALANCE<-1 AND DATA>ROOT->RIGHT->DATA
RETURN rotateLeft(ROOT)
IF BALANCE>1 AND DATA>ROOT->LEFT->DATA
ROOT->LEFT=rotateLeft(ROOT->LEFT)
RETURN rotateRight(ROOT)
IF BALANCE<-1 AND DATA<ROOT->RIGHT->DATA
ROOT->RIGHT=rotateRight(ROOT->RIGHT)
RETURN rotateLeft(ROOT)
[END OF IF]
Step 6: RETURN ROOT

## Algorithm to delete a node in AVL tree:

deleteNode (ROOT, DATA)
Step 1: IF ROOT = NULL, then
RETURN ROOT

```
        [END OF IF]
Step 2: IF DATA < ROOT->DATA
                ROOT->LEFT=deleteNode(ROOT->LEFT, DATA)
        ELSE IF DATA>ROOT->DATA
                ROOT->RIGHT=deleteNode(ROOT->RIGHT,DATA)
        ELSE
                IF ROOT->LEFT=NULL OR ROOT->RIGHT=NULL
                        SET TEMP=ROOT->LEFT ? ROOT->LEFT : ROOT->RIGHT
                        IF TEMP=NULL
                                TEMP=ROOT
                                ROOT=NULL
                        ELSE
                                ROOT=TEMP
                        [END OF IF]
                        FREE(TEMP)
                ELSE
                        TEMP=smallestNode(ROOT->RIGHT)
                        ROOT->DATA=TEMP->DATA
                        ROOT->RIGHT=deleteNode(ROOT->RIGHT, TEMP->DATA)
                [END OF IF]
        [END OF IF]
Step 3: IF (ROOT=NULL)
                RETURN ROOT
Step 4: ROOT->HEIGHT = 1 + MAX(height(ROOT->LEFT),height(ROOT->RIGHT))
Step 5: SET BALANCE = ROOT->LEFT->HEIGHT – ROOT->RIGHT->HEIGHT
Step 6: IF BALANCE>1 AND DATA < ROOT->LEFT->DATA
                RETURN rotateRight(ROOT)
        IF BALANCE<-1 AND DATA>ROOT->RIGHT->DATA
                RETURN rotateLeft(ROOT)
        IF BALANCE>1 AND DATA>ROOT->LEFT->DATA
                ROOT->LEFT=rotateLeft(ROOT->LEFT)
                RETURN rotateRight(ROOT)
        IF BALANCE<-1 AND DATARIGHT->DATA
                ROOT->RIGHT=rotateRight(ROOT->RIGHT)
                RETURN rotateLeft(ROOT)
        [END OF IF]
Step 7: RETURN ROOT
```

## Algorithm to search an element in AVL tree:

```
search (ROOT, VAL)
Step 1: IF ROOT ->DATA = VAL OR ROOT = NULL, then
                Return ROOT
        ELSE
                IF VAL < ROOT ->DATA
                        Return search(ROOT->LEFT,VAL)
```

```
            ELSE
                Return search(ROOT->RIGHT,VAL)
            [END OF IF]
        [END OF IF]
Step 2: EXIT
```

## Algorithm to find height/depth of AVL tree:

```
Height (ROOT)
Step 1: IF ROOT = NULL, then
            Return 0
        ELSE
            Return ROOT->HEIGHT
        [END OF IF]
Step 2: EXIT
```

## Algorithm to count total number of nodes in AVL tree:

```
totalNodes (ROOT)
Step 1: IF ROOT = NULL, then
            Return 0
        ELSE
            Return totalNodes(ROOT ->LEFT) + totalNodes(ROOT ->RIGHT) + 1
        [END OF IF]
Step 2: EXIT
```

## Algorithm to display the AVL tree:

```
displayTree (ROOT, space)
Step 1: IF (ROOT=NULL) then
            Goto step 8
        [END OF IF]
Step 2: SET space = space + 8
Step 3: displayTree (ROOT->RIGHT, space)
Step 4: SET I=1
Step 5: Repeat while I < space
            PRINT " "
            I=I+1
        [END OF LOOP]
Step 6: PRINT ROOT->DATA
Step 7: displayTree (ROOT->LEFT, space)
Step 8: EXIT
```

**DSA LAB**
**Lab Assignment number 14**

**Name:** Aamir Ansari **Batch:** A **Roll no:** 01

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

struct node {    // stucture of node
   struct node *left;
   int data;
   struct node *right;
   int height;
};

// declaring root
struct node *root = NULL;

struct node *findMax(struct node *root) {
   while (root->right != NULL) {
      root = root->right;
   }
   return root;
}

struct node *findMin(struct node *root) {
   while (root->left != NULL) {
      root = root->left;
   }
   return root;
}

int max (int n1, int n2) {
   return ((n1 > n2) ? n1 : n2);
}

int height (struct node *root) {
   if (root == NULL) {
      return 0;
   }
   return root->height;
}

struct node *getNewNode(int data) {    // initialises and allocates memory for newNode
   struct node *newNode;
   newNode = (struct node *)malloc(sizeof(struct node));
   newNode->data  = data;
   newNode->left  = NULL;
   newNode->right = NULL;
   newNode->height = 1;
   return newNode;
```

```
}

int getBalance(struct node *root) {
    if (root == NULL) {
        return 0;
    }
    return (height(root->left) - height(root->right));
}

struct node *rightRotate(struct node *root) {
    struct node *rootLeft = root->left;
    struct node *rootLeftRight = rootLeft->right;

    // rotation
    rootLeft->right = root;
    root->left = rootLeftRight;

    // updation of height
    root->height = max(height(root->left), height(root->right)) + 1;
    rootLeft->height = max(height(rootLeft->left), height(rootLeft->right)) + 1;

    // back tracking of root
    return rootLeft;
}

struct node *leftRotate(struct node *root) {
    struct node *rootRight = root->right;
    struct node *rootRightLeft = rootRight->left;

    // rotation
    rootRight->left = root;
    root->right = rootRightLeft;

    // updation of height
    root->height = max(height(root->left), height(root->right)) + 1;
    rootRight->height = max(height(rootRight->left), height(rootRight->right)) + 1;

    // back tracking of root
    return rootRight;
}

struct node *insert (struct node *root, int data) {    // inserts in the avl tree
    if (root == NULL) {    // base case
        root = getNewNode(data);
        return root;
    }
    if (data < root->data) {    // insertion in right sub-tree
        root->left = insert(root->left, data);
    }
    else if (data > root->data) {    // insertion in left sub-tree
        root->right = insert(root->right, data);
    }
```

```
    else {    // return root if value is equal
        return root;
    }

    // updating height of ancestor node
    root->height = max(height(root->left), height(root->right)) + 1;

    // balance factor
    int balance = getBalance(root);

    // ROTATIONS
    if ((balance > 1) && (data < root->left->data)) {    // LEFT-LEFT
        return rightRotate(root);
    }
    else if ((balance < -1) && (data > root->right->data)) {    // RIGHT-RIGHT
        return leftRotate(root);
    }
    else if ((balance > 1) && (data > root->left->data)) {    // LEFT-RIGHT
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }
    else if ((balance < -1) && (data < root->right->data)) {    // RIGHT-LEFT
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }
    return root;
}

struct node *delete(struct node *root, int val) {
    // deletion of node
    if (root == NULL) {    // empty tree
        return root;
    }
    else if (val < root->data) {    // finding node in left sub-tree
        root->left = delete (root->left, val);
    }
    else if (val > root->data) {    // finding node in right sub-tree
        root->right = delete (root->right, val);
    }
    else {    // found the node
        if (root->right == NULL && root->left == NULL) {    // deleting leaf node
            free(root);
            root = NULL;
        } else if (root->right == NULL) {    // deleting a node with only left sub-tree
            struct node *temp = root;
            root = root->left;
            free(temp);
        } else if (root->left == NULL) {    // deleting a node with only right sub-tree
            struct node *temp = root;
            root = root->right;
            free(temp);
        } else {    // deleting nodes with two sub-trees
```

```
            // storing address of node with min value in right sub-tree
            struct node *temp = findMin(root->right);
            root->data = temp->data;
            root->right = delete (root->right, temp->data);
        }
    }
    // updation of height
    root->height = max(height(root->left), height(root->right)) + 1;

    // check balance factor
    int balance = getBalance(root);

    // ROTATIONS
    if ((balance > 1) && (getBalance(root->left)>=0)) {    // LEFT-LEFT
        return rightRotate(root);
    }
    else if ((balance < -1) && (getBalance(root->right)<=0)) {    // RIGHT-RIGHT
        return leftRotate(root);
    }
    else if ((balance > 1) && (getBalance(root->left)<0)) {    // LEFT-RIGHT
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }
    else if ((balance < -1) && (getBalance(root->right)>0)) {    // RIGHT-LEFT
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }
    return root;
}

void search(struct node *root, int val) {
    if (root->data == val) {
        printf("\n%d is present in the tree", val);
        return;
    }
    if ((root->right == NULL && root->left == NULL) || root == NULL) {
        printf("\nNot present");
        return;
    }
    if (val <= root->data) {    // search in left sub-tree
        search(root->left, val);
    }
    else {    // search in right sub-tree
        search(root->right, val);
    }
}

int countAllNodes(struct node *root) {
    if (root == NULL) {
        return 0;
    }
    else {
```

```
        return countAllNodes(root->left) + countAllNodes(root->right) + 1;
    }
}

void inOrderTraversal (struct node *root) {
    if (root == NULL) {
        return;
    }
    inOrderTraversal(root->left);
    printf("%d  ", root->data);
    inOrderTraversal(root->right);
}

void display(struct node *root, int space) {
    if (root == NULL)
        return;

    // Increase distance between levels
    space += 7;

    // Process right child first
    display(root->right, space);

    // Print current node after space
    printf("\n");
    for (int i = 5; i < space; i++) {
        printf(" ");
    }
    printf("%d\n", root->data);

    // Process left child
    display(root->left, space);
}

int main() {
    struct node *temp;
    int data, i, choice, val;

    while (1) {
        printf("\n(1)  Insert");
        printf("\n(2)  Delete");
        printf("\n(3)  Search");
        printf("\n(4)  Height");
        printf("\n(5)  INORDER");
        printf("\n(6)  TOTAL number of nodes");
        printf("\n(7)  Display");
        printf("\n(8)  EXIT");
        printf("\nEnter your choice :  ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
```

```
            printf("\nEnter data to insert :  ");
            scanf("%d", &data);
            root = insert(root, data);
            printf("\n%d is inserted!", data);
            break;

        case 2:
            printf("\nEnter a value to delete :  ");
            scanf("%d", &val);
            root = delete (root, val);
            printf("\n%d is deleted!", val);
            break;

        case 3:
            printf("\nEnter a number to Search");
            scanf("%d", &data);
            search(root, data);
            break;

        case 4:
            printf("\nHeight of tree is :  %d", height(root));
            break;

        case 5:
            printf("\nIN-ORDER :  ");
            inOrderTraversal(root);
            break;

        case 6:
            printf("\nTotal number of nodes :  %d", countAllNodes(root));
            break;

        case 7:
            display(root, 0);
            break;

        case 8:
            printf("\n*** E X I T I N G ***\n");
            exit(1);
            break;

        default:
            printf("\n*** I N V A L I D ***");
      }
   }
   return 0;
}
```

// output

```
(1)  Insert
(2)  Delete
(3)  Search
(4)  Height
(5)  INORDER
(6)  TOTAL number of nodes
(7)  Display
(8)  EXIT
Enter your choice :  1

Enter data to insert :  30

30 is inserted!
(1)  Insert
(2)  Delete
(3)  Search
(4)  Height
(5)  INORDER
(6)  TOTAL number of nodes
(7)  Display
(8)  EXIT
Enter your choice :  1

Enter data to insert :  20

20 is inserted!
(1)  Insert
(2)  Delete
(3)  Search
(4)  Height
(5)  INORDER
(6)  TOTAL number of nodes
(7)  Display
(8)  EXIT
Enter your choice :  1

Enter data to insert :  10

10 is inserted!
(1)  Insert
(2)  Delete
(3)  Search
(4)  Height
(5)  INORDER
(6)  TOTAL number of nodes
(7)  Display
(8)  EXIT
Enter your choice :  7
```

```
      -

         30

  20

         10

(1)  Insert
(2)  Delete
(3)  Search
(4)  Height
(5)  INORDER
(6)  TOTAL number of nodes
(7)  Display
(8)  EXIT
Enter your choice :  3

Enter a number to Search30

30 is present in the tree
(1)  Insert
(2)  Delete
(3)  Search
(4)  Height
(5)  INORDER
(6)  TOTAL number of nodes
(7)  Display
(8)  EXIT
Enter your choice :  4

Height of tree is :  2
(1)  Insert
(2)  Delete
(3)  Search
(4)  Height
(5)  INORDER
(6)  TOTAL number of nodes
(7)  Display
(8)  EXIT
Enter your choice :  6

Total number of nodes :  3
(1)  Insert
(2)  Delete
(3)  Search
(4)  Height
(5)  INORDER
(6)  TOTAL number of nodes
(7)  Display
(8)  EXIT
Enter your choice :  8

*** E X I T I N G ***
```

**DSA LAB**
**Lab Assignment number 15**

**Name:** Aamir Ansari **Batch:** A **Roll no:** 01

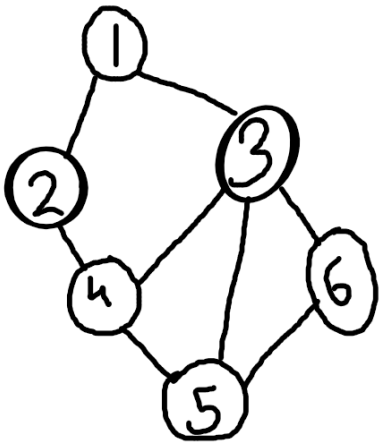
**Aim:** Implementation of BFS and DFS on a directed graph using adjacency matrix.

## A) BFS Algorithm:

BFS is Breadth First Search

1. Select any random node
2. Add the selected node to queue and list of BFS traversal
3. Add all nodes of selected node from adjacency matrix to the queue and the list of BFS traversal
4. After adding all the nodes from adjacency list, delete from rear in the queue
5. Consider rear node and add all the adjacent node
6. Repeat the process till all nodes are visited

.




Consider the following graph, it's BFS traversal would be

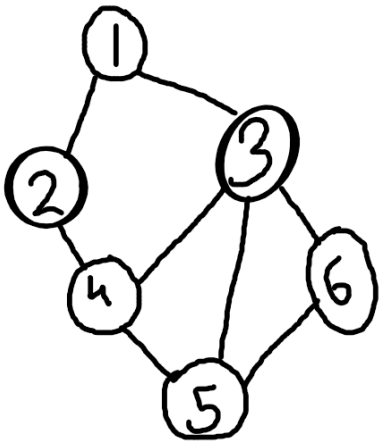
1, 2, 3, 4, 5, 6

## B) DFS Algorithm

DFS is Depth Frst Search tree

1. Select any random node
2. Push the node to the stack and to the list of DFS traversal
3. Go to any one of the node in the adjacency list of the selected node
4. Repeat this process till all the connections are visited
5. Then pop from the stack
6. Trace the other node in the adjacency list of the top node
7. Repeat till the stack is empty

.




Consider the above graph, it's DFS traversal would be:

1, 2, 4, 3, 6, 5

## Implementation of BSF

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

int adjacancy[10][10];
int queue[20], front = 0, rear = -1;
int visited[20];
int n, i, j;

void bfs(int startVertex) {
   for (i = 1; i <= n; i++) {
      if (adjacancy[startVertex][i] && !visited[i]) {
         queue[++rear] = i;
      }
   }
   if (front <= rear) {
      visited[queue[front]] = 1;
      bfs(queue[front++]);
   }
}

int main() {
   printf("Enter number of vertices :  ");
   scanf("%d", &n);

   for (i = 1; i <= n; i++) {    // intialisation
      queue[i] = 0;
      visited[i] = 0;
   }

   // input of adjacancy matrix
   printf("\nEnter adjacancy matrix :  \n");
   for (i = 1; i <= n; i++) {
      for (j = 1; j <= n; j++) {
         scanf("%d", &adjacancy[i][j]);
      }
   }
   bfs(1);

   printf("\nBFS traversal is :  ");
   for (i = 1; i <= n; i++) {
      if (visited[i]) {
         printf("%d   ", i);
      }
      else {
         printf("\nEnetered graph is incorrect, since all nodes are not reachable!");
         break;
      }
   }
}
```

// output

```
Enter number of vertices :  5

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

## Implementation of DFS

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

int n;
void dfs(int adj[][n],int visited[],int start) {
   int stack[n];
   int top = -1, i;
   printf("%c->  ",start+65);
   visited[start] = 1;
   stack[++top] = start;
   while(top != -1) {
      start = stack[top];
      for(i = 0; i < n; i++) {
         if(adj[start][i] && visited[i] == 0) {
            stack[++top] = i;
            printf("%c->  ", i+65);
            visited[i] = 1;
            break;
         }
      }
      if (i == n) {
         top--;
      }
   }
}

int main() {
   int adj[10][10];
   int visited[20] = {0}, i, j;

   printf("Enter number of vertices :  ");
   scanf("%d", &n);

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

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

// output
  Enter number of vertices :  5

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

**DSA LAB**

**Lab Assignment number 16**

**Name:** Aamir Ansari **Batch:** A **Roll no:** 01

**AIM:** To implement Binary Search

## ALGORITHM:

Step 1: READ n and elements of list
Step 2: [INITIALIZE] first = 0
Step 3: [INITIALIZE] last = n - 1
Step 4: [INITIALIZE] middle = (first+last)/2
Step 5:Repeat the following while first <= last
IF array[middle] < search
SET first = middle + 1
ELSE IF array[middle] == search
PRINT “Found”
break
ELSE
SET last = middle - 1
SET middle = (first + last)/2
Step 6: IF first > last
PRINT "Not found”
Step 7:EXIT

## EXAMPLE:

array[5] = { 11, 12,13,14,15}
Search : 14
Stage 0 :
First :0
Last :4
Middle : 2

Search > array[middle]
Stage 1:
First : 3
Last :4
Middle : 3

Search found at location 3

**DSA LAB**

**Lab Assignment number 16**

**Name:** Aamir Ansari **Batch:** A **Roll no:** 01

**Aim:** Implementation of Binary Search

## Program:

```
#include <stdio.h>

/*Array to store the list*/
int array[100];

/*Function to perform Binary Search*/
void binary_search(int n, int search)
{
   int first ,last, middle;
   first = 0;
   last = n - 1;
   middle = (first+last)/2;
   while (first <= last)
   {
      if (array[middle] < search)
      {
         first = middle + 1;
      }
      else if (array[middle] == search)
      {
         printf("%d found at location %d.\n", search, middle+1);
         break;
      }
      else
      {
         last = middle - 1;
      }
      middle = (first + last)/2;
   }
   if (first > last)
   {
      printf("Not found! %d is not present in the list.\n", search);
   }
}
void main()
{
   int c,n,search;
   printf("BINARY SEARCH\n");
   printf("Enter number of elements in list : ");
   scanf("%d", &n);
   printf("Enter %d integers\n", n);
   /*Taking the inputs*/
```

```
    for (c = 0; c < n; c++)
    {
       scanf("%d", &array[c]);
    }

    printf("Enter element to search : ");
    scanf("%d", &search);
    binary_search(n,search);
}
```

## Output:

```
BINARY SEARCH
Enter number of elements in list : 5
Enter 5 integers
11
12
13
14
15
Enter element to search : 13
13 found at location 3.
```

# DSA LAB

## Lab Assignment number 17

**Name:** Aamir Ansari **Batch:** A **Roll no:** 01

**AIM:** To implement Selection sort and Insertion sort

## ALGORITHM:

Selection Sort:

```
S 1: READ n and Elements in array
S 2: [INITIALIZE] i=0
S 3: Repeat  following while i<n-1
        SET min = i
        [INITIALIZE] j=i+1
        Repeat following while j<n
                IF array[j] < array[min]
                        SET min = j
                SET j++
        swap(&array[min], &array[i])
        SET i++
S 4: EXIT
swap(int *a, int *b):
S 1: SET temp = *a
S 2: SET *a = *b
S 3: SET *b = temp
S 4: EXIT
```

Insertion Sort:

```
S 1: READ n and Elements in array
S 2: [INITIALIZE] i=1
S 3: Repeat  following while i<=n-1
        SET temp = array[i]
        [INITIALIZE] j=i-1
        Repeat following while j>=0
                IF array[j] > temp
                        SET array[j+1] = array[j]
                        SET flag = 1
                ELSE
                        break
                SET j--
        IF flag == 1
                SET array[j+1] = temp
        SET i++
S 4: EXIT
```

**EXAMPLE:**

array[5] = { 17 , 5 ,67 , 45, 22}

Selection Sort:

For i=0;  5, 17 , 67 , 45 , 22

For i=1; 5, 17 , 67 , 45 , 22

For i=2; 5, 17 , 22 , 45 , 67

For i=3;  5, 17 , 22 , 45 , 67

For i=4;  5, 17 , 22 , 45 , 67

Sorted Array : 5, 17 , 22 , 45 , 67

Insertion Sort:

For i=1;  5, 17 , 67 , 45 , 22

For i=2;  5, 17 , 67 , 45 , 22

For i=3;  5, 17 , 45 , 67 , 22

For i=4;  5, 17 , 22 , 45 , 67

Sorted Array : 5, 17 , 22 , 45 , 67

**DSA LAB**

**Lab Assignment number 17**

**Name:** Aamir Ansari **Batch:** A **Roll no:** 01

**Aim:** To implement Selection sort and Insertion sort

## Program:

```
#include <stdio.h>

/*Array to store the list*/
int array[100];

/*Function to swap */
void swap(int *a, int *b)
{
   int temp = *a;
   *a = *b;
   *b = temp;
}

/*Insertion Sort*/
void insertion_sort(int n)
{
   int i,j,temp,flag;
   for (i = 1 ; i <= n - 1; i++)
   {
      temp = array[i];
      for (j = i - 1 ; j >= 0; j--)
      {
         if (array[j] > temp)
         {
            array[j+1] = array[j];
            flag = 1;
         }
         else
         {
            break;
         }
      }
      if (flag)
      {
         array[j+1] = temp;
      }
   }
}

/*Selection Sort*/
void selection_sort(int n)
{
```

```
    int i, j, min;

    for (i = 0; i < n-1; i++)
    {
        min = i;
        for (j = i+1; j < n; j++)
        {
            if (array[j] < array[min])
            {
                min = j;
            }
        }
        swap(&array[min], &array[i]);
    }
}

/*Print a sorted array*/
void print_sorted_array(int n)
{
    int i;
    printf("Sorted Array:");
    for(i=0;i<n;i++)
    {
        printf("%d ",array[i]);
    }
}

int main()
{
    int n,i, choice;

    printf("Enter number of elements\n");
    scanf("%d", &n);

    printf("Enter %d integers\n", n);

    for (i = 0; i < n; i++)
    {
        scanf("%d", &array[i]);
    }

    printf("Type of sort to perform:\n1.Selection Sort\n2.Insertion Sort\n3.Exit");
    printf("Enter the choice to be performed: ");
    scanf("%d",&choice);

    switch(choice)
    {
        case 1:
            selection_sort(n);
            print_sorted_array(n);
            break;
```

```
        case 2:
            insertion_sort(n);
            print_sorted_array(n);
            break;

        case 3:
        default:
            printf("Thank You!!");
    }

    return 0;
}
```

## Output:

```
Enter number of elements
5
Enter 5 integers
12
34
2
17
8
Type of sort to perform:
1.Selection Sort
2.Insertion Sort
3.ExitEnter the choice to be performed: 1
Sorted Array:2 8 12 17 34
```

```
Enter number of elements
5
Enter 5 integers
22
98
56
3
11
Type of sort to perform:
1.Selection Sort
2.Insertion Sort
3.ExitEnter the choice to be performed: 2
Sorted Array:3 11 22 56 98
```

**DSA LAB**

**Lab Assignment number 18**

**Name:** Aamir Ansari **Batch:** A **Roll no:** 01

**Aim:** Implement Merge Sort and Quick Sort

## Program:

```
#include<stdio.h>

/*Array to store the list*/
int array[1000];

/*Merge Sort*/
void merge(int first, int last)
{
   int mid = (first+last)/2;
   int i = first;
   int j = mid+1;
   int k = first;

   int temp[100];

   while(i<=mid && j<=last)
   {
      if(array[i] < array[j])
      {
         temp[k++] = array[i++];
      }
      else
      {
         temp[k++] = array[j++];
      }
   }

   while(i<=mid)
   {
      temp[k++] = array[i++];
   }

   while(j<=last)
   {
      temp[k++]  = array[j++];
   }

   /*Copy all element to original array*/
   for(int i=first;i<=last;i++)
   {
      array[i] = temp[i];
```

```
    }
}

void merge_sort(int first,int last)
{
    /*Base case - 1 or 0 elements*/
    if(first>=last)
    {
        return;
    }
    int mid = (first+last)/2; /*Divide*/

    /*Recursively the arrays - first,mid and mid+1,last*/
    merge_sort(first,mid);
    merge_sort(mid+1,last);

    /*Merge the two parts*/
    merge(first,last);

}

/*Quick Sort*/
void quick_sort(int first,int last)
{
    int i, j, pivot, temp;

    if(first<last)
    {
        pivot=first;
        i=first;
        j=last;

        while(i<j)
        {
            while(array[i]<=array[pivot]&&i<last)
            {
                i++;
            }
            while(array[j]>array[pivot])
            {
                j--;
            }
            if(i<j)
            {
                temp=array[i];
                array[i]=array[j];
                array[j]=temp;
            }
        }
        temp=array[pivot];
        array[pivot]=array[j];
        array[j]=temp;
```

```
        quick_sort(first,j-1);
        quick_sort(j+1,last);
     }
}

/*Print a sorted array*/
void print_sorted_array(int n)
{
   int i;
   printf("Sorted Array:");
   for(i=0;i<n;i++)
   {
      printf("%d ",array[i]);
   }
}

void main()
{
   int n,i, choice;

   printf("Enter number of elements in the List : ");
   scanf("%d", &n);
   printf("Enter %d integers\n", n);
   for (i = 0; i < n; i++)
   {
      scanf("%d", &array[i]);
   }

   printf("Type of sort to perform:\n1Merge Sort\n2.Quick Sort\n3.Exit");
   printf("\nEnter the choice to be performed: ");
   scanf("%d",&choice);

   switch(choice)
   {
      case 1:
         merge_sort(0,n-1);
         print_sorted_array(n);
         break;

      case 2:
         quick_sort(0,n-1);
         print_sorted_array(n);
         break;

      case 3:
      default:
         printf("Thank You!!");
   }
}
```

**Output:**

```
Enter number of elements in the List : 6
Enter 6 integers
1
30
75
16
55
28
Type of sort to perform:
1Merge Sort
2.Quick Sort
3.Exit
Enter the choice to be performed: 1
Sorted Array:1 16 28 30 55 75
```

```
Enter number of elements in the List : 5
Enter 5 integers
34
98
65
11
3
Type of sort to perform:
1Merge Sort
2.Quick Sort
3.Exit
Enter the choice to be performed: 2
Sorted Array:3 11 34 65 98
```

**DSA LAB**

**Lab Assignment number 18**

**Name:** Aamir Ansari **Batch:** A **Roll no:** 01

**Aim:** To implement Merge Sort and Quick Sort

## Algorithm:

Merge Sort:

```
Step 1: READ n
Step 2: [INITIALIZE] first =0 ,last = n-1
Step 3: IF first>=last
        RETURN
Step 4: [INITIALIZE] mid = (first+last)/2
Step 5: merge_sort(first,mid)
Step 6: merge_sort(mid+1,last)
Step 7: merge(first,last)
Step 8:EXIT
Merge :
Step 1: [INITIALIZE] mid = (first+last)/2
Step 2: [INITIALIZE] i = first , j = mid+1 , k = first
Step 3: [INITIALIZE] temp[100]
Step 4: Repeat following while i<=mid AND  j<=last
        IF array[i] < array[j]
                SET temp[k++] = array[i++]
        ELSE
                SET temp[k++] = array[j++]
Step 5: Repeat following while i<=mid
        SET temp[k++] = array[i++]
Step 6: Repeat following while j<=last
        SET temp[k++]  = array[j++]
Step 7: [INITIALIZE] i = first
Step 8: Repeat while i<=last
        SET array[i] = temp[i]
        SET i++
Step 9: EXIT
```

Quick Sort:

```
Step 1: READ n
Step 2: [INITIALIZE] first =0 ,last = n-1
Step 3: IF   first<last
        [INITIALIZE] pivot=first , i=first , j=last
        Repeat following while i<j
                Repeat following while array[i]<=array[pivot] AND i<last
                        SET i++
                Repeat following while array[j]>array[pivot]
                        SET  j--
                IF i<j
                        SET temp=array[i]
                        SET array[i]=array[j]
```

SET array[j]=temp
SET temp=array[pivot]
SET array[pivot]=array[j]
SET array[j]=temp

quick_sort(first,j-1)
quick_sort(j+1,last) [END OF IF]
Step 4:EXIT

## EXAMPLE:

Array[6] = {1,45,76,22,38,99}

Merge Sort:
[1,45,76,22,38,99]
Dividing and calling merge_sort(1):
([1 ,45,76] [22,38,99])
Dividing and calling merge_sort(2):
(([1,45] [76]) ([22,38][99]))
Dividing and calling merge_sort(3):
((([1][45])[76])(([22][38])[99]))
Calling merge(1):
(([1,45][76])([22,38][99]))
Calling merge(2):
(([1,45,76])([22,38,99]))
Calling merge(3):
[1,22,38,45,76,99]
Sorted array: 1,22,38,45,76,99

Quick Sort:
[1,45,76,22,38,99]
Pivot: 99
[1,45,76,22,38] [99]
Pivot: 38
[1,22,][38] [45,76] [99]
Pivot: 22 Pivot : 76
[1][22][38][45][76][99]
Sorted array: 1,22,38,45,76,99

# DSA
# Assignment 01

**Name:** Aamir Ansari **Batch:** A **Roll no:** 01

### Assignment on Dijkstra Algorithm to find shortest path

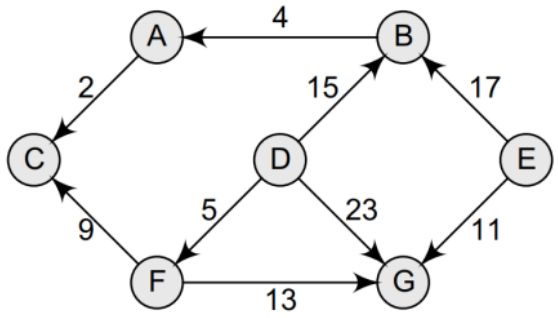
Dijkstra's algorithm is used to find the length of an optimal path between two nodes in a graph. The term optimal can mean anything, shortest, cheapest, or fastest. If we start the algorithm with an initial node, then the distance of a node Y can be given as the distance from the initial node to that node.

**Algorithm for Dijkstra's algorithm:**

Step 1: Select the source node also called the initial node
Step 2: Define an empty set N that will be used to hold nodes to which a shortest path has been found.
Step 3: Label the initial node with, and insert it into N.
Step 4: Repeat Steps 5 to 7 until the destination node is in N or there are no more labelled nodes in N.
Step 5: Consider each node that is not in N and is connected by an edge from the newly inserted node.
Step 6: (a) If the node that is not in N has no label then SET the label of the node = the label of the newly inserted node + the length of the edge.
(b) Else if the node that is not in N was already labelled, then SET its new label = minimum (label of newly inserted vertex + length of edge, old label)
Step 7: Pick a node not in N that has the smallest label assigned to it and add it to N.

**Example:**

Consider the graph G given alongside. Taking D as the initial node, we execute the Dijkstra's algorithm.
Step 1: Set the label of D = 0 and N = {D}.
Step 2: Label of D = 0, B = 15, G = 23, and F = 5.
Therefore, N = {D, F}.
Step 3: Label of D = 0, B = 15, G has been relabelled 18 because minimum (5 + 13, 23) = 18, C has been relabeled 14 (5 + 9). Therefore, N = {D, F, C}.
Step 4: Label of D = 0, B = 15, G = 18. Therefore, N = {D, F, C, B}.
Step 5: Label of D = 0, B = 15, G = 18 and A = 19 (15 + 4). Therefore, N = {D, F, C, B, G}.
Step 6: Label of D = 0 and A = 19. Therefore, N = {D, F, C, B, G, A}



# DSA
# Assignment 02

**Name:** Aamir Ansari **Batch:** A **Roll no:** 01

**Assignment on Recursion and Storage management**

## Recursion

Recursion is a technique that breaks a problem into one or more sub-problems that are similar to the original problem. A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself. Every recursive solution has two major cases.
They are

1. **Base case**, in which the problem is simple enough to be solved directly without making any further calls to the same function.
2. **Recursive case**, in which first the problem at hand is divided into simpler sub-parts. Second the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts

## Types of Recursion

1. Direct Recursion: A function is said to be directly recursive if it explicitly calls itself.
2. Indirect Recursion: A function is said to be indirectly recursive if it contains a call to an other function which ultimately calls it.
3. Tail Recursion: A recursive function is said to be tail recursive if no operations are pending to be performed when the recursive function returns to its caller. when the called function returns, the returned value is immediately returned from the calling function. Tail recursive functions are highly desirable because they are much more efficient to use as the amount of information that has to be stored on the system stack is independent of the number of recursive calls.

## Sequential Fit Methods

### First Fit

In the first fit approach is to allocate the first free partition or hole large enough which can accommodate the process. It finishes after finding the first suitable free partition. This method keeps the free/busy list of jobs organized by memory location, low-ordered to high-ordered memory. The o erating system doesn't search for appropriate partition but just allocate the job to the nearest memory partition available with sufficient size.

Advantages:

It is fast in processing. As the processor allocates the nearest available memory partition to the job, it is very fast in execution.

Disadvantages:

It wastes a lot of memory. The processor ignores if the size of partition allocated to the job is very large as compared to the size of job or not. It just allocates the memory. As a result, a lot of memory

is wasted and many jobs may not get space in the memory, and would have to wait for another job to complete.

**Best Fit**

The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This method keeps the free/busy list in order by size – smallest to largest. In this method, the operating system first searches the whole of the memory according to the size of the given job and allocates it to the closest-fitting free partition in the memory, making it able to use memory efficiently.

Advantages:

It is memory efficient. The operating system allocates the job minimum possible space in the memory, making memory management very efficient. To save memory from getting wasted, it is the best method.

Disadvantages:

It is a slow process. Checking the whole memory for each job makes the working of the operating system very slow. It takes a lot of time to complete the work.

**Worst fit**

In this allocation technique the process traverses the whole memory and always search for largest hole/partition, and then the process is placed in that hole/partition. It is a slow process because it has to traverse the entire memory to search largest hole.

Advantages:

Since this process chooses the largest hole/partition, therefore there will be large internal fragmentation. Now, this internal fragmentation will be quite big so that other small processes can also be placed in that left over partition.

Disadvantages:

It is a slow process because it traverses all the partitions in the memory and then selects the largest partition among all the partitions, which is a time-consuming process.

**Fragmentation**

Fragmentation is an unwanted problem that deals with memory fragment and occur due to either non-contiguous memory allocation or fixed size memory allocation. As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.

**Types of Fragmentation**

There are two types of fragmentation:

**Internal Fragmentation:**

Internal fragmentation happens when the memory is split into mounted sized blocks. Whenever a method requests for the memory, the mounted sized block is allotted to the method. just in case the memory allotted to the method is somewhat larger than the memory requested, then the distinction between allotted and requested memory is that the Internal fragmentation.

**External Fragmentation:**

External fragmentation happens when there's a sufficient quantity of area within the memory to satisfy the memory request of a method. However, the process's memory request cannot be fulfilled because the memory offered is during a non-contiguous manner. Either you apply first-fit or best-fit memory allocation strategy it'll cause external fragmentation.