RUSTAMJI INSTITUTE OF TECHNOLOGY

BSF ACADEMY, TEKANPUR

Lab File for

CS303 (Data Structure)

Submitted by

PRIYANKA (0902CS231075)
B.Tech. Computer Science & Engineering 3rd Semester
(2023-2027 batch)

Subject Teacher
Dr. Jagdish Makhijani

File Checked by
Mr. Yashwant Pathak

Self-Declaration Certificate

I, **Priyanka**, hereby declare that I have completed the lab work of CS303 (Data Structure) at my own effort and understanding.

I affirm that the work submitted is my own, and I take full responsibility for its authenticity and originality.

Date:

Priyanka

0902CS231075

ENVORIONMENT USED

Hardware Configuration :

Processor : Intel(R) Core(TM) i5 CPU M 430 @ 2.27GHz 2.26GHz

C Compiler : GCC Compiler

User Interface : <Visual Studio Code>

GROUP MEMBERS

Member-1 : Priyanka (0902CS231075)

Member-2 : Saloni Singh Gour(0902CS231092)

Member-3 : Shreya Gautam(0902CS231108)

Member-4 : Shivam Sharma(0902CS231107)

TABLE OF CONTENTS

Section-A (Linked List)

Section-B (Stack)

Section-C (Queue)

Section-D (Trees)

S. No.	Practical Description	Page Nos.	Cos
1	Implementation of Binary Search Tree.	36-39	CO-3
2	Conversion of BST PreOrder/PostOrder/InOrder.	40-41	CO-3
3	Implementation of Kruskal Algorithm	42-44	CO-3
4	Implementation of Prim Algorithm	45-47	CO-3
5	Implementation of Dijkstra Algorithm	48-49	CO-3

Section-E (Sorting & Searching)

S. No.	Practical Description	Page Nos.	Cos
1	Implementation of Sorting a. Bubble b. Selection c. Insertion d. Quick e. Merge	50-60	CO-5
2	Implementation of Binary Search on a list of numbers stored in an Array	61-62	CO-5
3	Implementation of Binary Search on a list of strings stored in an Array	63-64	CO-5
4	Implementation of Linear Search on a list of strings stored in an Array	65	CO-5
5	Implementation of Binary Search on a list of strings stored in a Single Linked List	66-67	CO-5

Section-A (Linked List)

Experiment No.: 1

Program Description:

Implementation of Linked List using array.

Solution:

```
#include <stdio.h>
#include <stdlib.h>
//linked list with array
#define MAX_SIZE 100
struct Node {
    int data;
    int next;
};
struct LinkedList {
    struct Node array[MAX_SIZE];
    int head;
    int freeList;
};
void initialize(struct LinkedList *list) {
    list->head = -1;
    list->freeList = 0;
    for (int i = 0; i < MAX_SIZE - 1; ++i) {
        list->array[i].next = i + 1;
    }
  list->array[MAX_SIZE - 1].next = -1;
}
void insertAtBeginning(struct LinkedList *list, int data) {
    if (list->freeList == -1) {
        printf("Linked list is full. Cannot insert more
elements.\n");
        return;
    }
```

```
    int newIndex = list->freeList;
    list->freeList = list->array[newIndex].next;
    list->array[newIndex].data = data;
    list->array[newIndex].next = list->head;
    list->head = newIndex;
}
void display(struct LinkedList *list) {
    int current = list->head;
    printf("Linked List: ");
    while (current != -1) {
        printf("%d -> ", list->array[current].data);
        current = list->array[current].next;
    }
    printf("NULL\n");
}
int main() {
    struct LinkedList myList;
    initialize(&myList);
    insertAtBeginning(&myList, 3);
    insertAtBeginning(&myList, 7);
    insertAtBeginning(&myList, 1);
    display(&myList);
    return 0;
}
```

Output:

```
Linked List: 1 -> 7 -> 3 -> NULL
```

Experiment No.: 2

Program Description:

Implementation of Linked List using Pointers.

Solution:

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
  int data;
  struct node *next;
};
struct node *head;

// function prototyping

void insert_begin();
void add_last();
void display();

int main()
{
  insert_begin();
  insert_begin();
  add_last();

  display();
  return 0;
}

void display()
{
  struct node *temp;
  temp = head;
  printf("\nHead->");
  while (temp != NULL)
  {
    printf("%d->", temp->data);
    temp = temp->next;
  }
```

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

void insert_begin()
{
  struct node *newnode;
  int data;
  printf("enter the data for insertion\n");
  scanf("%d", &data);
  newnode = malloc(sizeof(struct node));

  if (newnode == NULL)
  {
    printf("memory allocation is not successful for insertion\n");
    return;
  }
  else
  {
    newnode->data = data;
    newnode->next = head;
    head = newnode;

    printf("insertion is successful\n");
  }
}

void add_last()
{
  struct node *newNode, *lastnode;
  int data;
  printf("enter the data for newNode which is going to insert at the
last\n");
  scanf("%d", &data);
  newNode = (struct node *)malloc(sizeof(struct node));
  newNode->data = data;
  newNode->next = NULL;

  if (head == NULL)
  {
    head = newNode;
  }
  else
```

```
  {
    lastnode = head;
    while (lastnode->next != NULL)
    {
      lastnode = lastnode->next;
    }
    lastnode->next = newNode;
    printf("\ninsertion is successfull at last of linked list\n");
  }
}
```

Output:

```
enter the data for insertion
10
insertion is successful
enter the data for insertion
20
insertion is successful
enter the data for newNode which is going to insert at the last
30

insertion is successfull at last of linked list

Head->20->10->30->Null
```

Experiment No.: 3

Program Description:

Implementation of Doubly Linked List using Pointers.

Solution:

```
#include <stdio.h>
#include <stdlib.h>
struct node
  {
    struct node *prev;
    struct node *next;
    int data;
  };
  struct node *head;
  void insertionFirst();
  void insertionLast();
  void insertionLoc();
  void deleteFirst();
  void deleteLast();
  void deleteLoc();
  void printList();
  void searchList();
  int main()
  {
    int choice = 0;
    while (choice != 9)
    {
      printf("\n\nDoubly Linked List Menu\n");
      printf("\n1.Insert at begining\n");
      printf("2.Insert at last\n");
      printf("3.Insert at any random location\n");
      printf("4.Delete from Beginning\n");
      printf("5.Delete from last\n");
      printf("6.Delete the node after the given data\n");
      printf("7.Search\n");
      printf("8.Show\n");
      printf("9.Exit\n");
      printf("\nEnter your choice?\n");
      scanf("\n%d", &choice);
```

```
    switch (choice)
    {
    case 1:
      insertionFirst();
      break;
    case 2:
      insertionLast();
      break;
    case 3:
      insertionLoc();
      break;
    case 4:
      deleteFirst();
      break;
    case 5:
      deleteLast();
      break;
    case 6:
      deleteLoc();
      break;
    case 7:
      searchList();
      break;
    case 8:
      printList();
      break;
    case 9:
      exit(0);
      break;
    default:
      printf("Invalid Choice!!! Please try again....");
    }
  }
  return 0;
}
void insertionFirst()
{
  struct node *ptr;
  int item;
  ptr = (struct node *)malloc(sizeof(struct node));
  if (ptr == NULL)
  {
```

```
    printf("\nOVERFLOW!!!");
  }
  else
  {
    printf("\nEnter value to insert: ");
    scanf("%d", &item);
    if (head == NULL)
    {
      ptr->next = NULL;
      ptr->prev = NULL;
      ptr->data = item;
      head = ptr;
    }
    else
    {
      ptr->data = item;
      ptr->prev = NULL;
      ptr->next = head;
      head->prev = ptr;
      head = ptr;
    }
    printf("\nNode inserted successfully....\n");
  }
}

void insertionLast()
{
  struct node *ptr, *temp;
  int item;
  ptr = (struct node *)malloc(sizeof(struct node));
  if (ptr == NULL)
  {
    printf("\nOVERFLOW!!!");
  }
  else

    printf("\nEnter value to insert: ");
  scanf("%d", &item);
  ptr->data = item;
  if (head == NULL)
  {
    ptr->next = NULL;
```

```
      ptr->prev = NULL;
      head = ptr;
    }
    else
    {
      temp = head;
      while (temp->next != NULL)
      {
        temp = temp->next;
      }
      temp->next = ptr;
      ptr->prev = temp;
      ptr->next = NULL;
    }
    printf("\nNode inserted successfully\n");
  }

void insertionLoc()
{
  struct node *ptr, *temp;
  int item, loc, i;
  ptr = (struct node *)malloc(sizeof(struct node));
  if (ptr == NULL)
  {
    printf("\n OVERFLOW!!!");
  }
  else
  {
    temp = head;
    printf("Enter the location (Starting Location is Zero): ");
    scanf("%d", &loc);
    for (i = 0; i < loc; i++)
    {
      temp = temp->next;
      if (temp == NULL)
      {
        printf("\nThere are less than %d elements\n", loc);
        return;
      }
    }
    printf("Enter value: ");
    scanf("%d", &item);
```

```
    ptr->data = item;
    ptr->next = temp->next;
    ptr->prev = temp;
    temp->next = ptr;
    temp->next->prev = ptr;
    printf("\nNode inserted successfully...\n");
  }
}

void deleteFirst()
{
  struct node *ptr;
  if (head == NULL)
  {
    printf("\nUNDERFLOW!!!");
  }
  else if (head->next == NULL)
  {
    head = NULL;
    free(head);
    printf("\nNode deleted successfully....\n");
  }
  else
  {
    ptr = head;
    head = head->next;
    head->prev = NULL;
    free(ptr);
    printf("\nNode deleted successfully....\n");
  }
}

void deleteLast()
{
  struct node *ptr;
  if (head == NULL)
  {
    printf("\nUNDERFLOW!!!");
  }
  else if (head->next == NULL)
  {
    head = NULL;
```

```
    free(head);
    printf("\nNode deleted successfully...\n");
  }
  else
  {
    ptr = head;
    if (ptr->next != NULL)
    {
      ptr = ptr->next;
    }
    ptr->prev->next = NULL;
    free(ptr);
    printf("\nNode deleted successfully...\n");
  }
}

void deleteLoc()
{
  struct node *ptr, *temp;
  int val;
  printf("\nEnter the data after which the node is to be deleted :
");
  scanf("%d", &val);
  ptr = head;
  while (ptr->data != val)
    ptr = ptr->next;
  if (ptr->next == NULL)
  {
    printf("\nCan't delete....\n");
  }
  else if (ptr->next->next == NULL)
  {
    ptr->next = NULL;
  }
  else
  {
    temp = ptr->next;
    ptr->next = temp->next;
    temp->next->prev = ptr;
    free(temp);
    printf("\nNode deleted successfully...\n");
  }
```

```
}

void printList()
{
  struct node *ptr;
  printf("\nThe Doubly Linked List is\nSTART -> ");
  ptr = head;
  while (ptr != NULL)
  {
    printf("%d -> ", ptr->data);
    ptr = ptr->next;
  }
  printf("NULL\n\n");
}

void searchList()
{
  struct node *ptr;
  int item, i = 0, flag;
  ptr = head;
  if (ptr == NULL)
    printf("\nEmpty List\n");
  else
  {
    printf("\nEnter item which you want to search?\n");
    scanf("%d", &item);
    while (ptr != NULL)
    {
      if (ptr->data == item)
      {
        printf("\nItem %d found at location %d ", item, i + 1);
        flag = 0;
        break;
      }
      else
      {
```

```
        flag = 1;
      }
      i++;
      ptr = ptr->next;
    }
    if (flag == 1)
    {
      printf("\nItem %d not found\n", item);
    }
  }
}
```

Output:

```
Doubly Linked List Menu

1.Insert at begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete the node after the given data

7.Search

8.Show

9.Exit

Enter your choice?

1

Enter value to insert: 10

Node inserted successfully....
```

Experiment No.: 4

Program Description:

Implementation of Circular Single Linked List using Pointers.

Solution:

Input:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *next;
};
struct node *head;

void beginsert ();
void lastinsert ();
void display();
void main ()
{
    int choice =0;
    while(choice != 7)
    {
        printf("\n***Main Menu***\n");
        printf("\nChoose one option from the following list ...\n");

printf("\n===============================================\n");
        printf("\n1.Insert in begining\n2.Insert at
last\n3.Display\n4.Exit\n");
```

```
        printf("\nEnter your choice?\n");
        scanf("\n%d",&choice);
        switch(choice)
        {
            case 1:
            beginsert();
            break;
            case 2:
            lastinsert();
            break;
            case 3:
            display();
            break;
            case 4:
            exit(0);
            break;
            default:
            printf("Please enter valid choice..");
        }
    }
}
void beginsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node *)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter the node data?");
```

```
        scanf("%d",&item);
        ptr -> data = item;
        if(head == NULL)
        {
            head = ptr;
            ptr -> next = head;
        }
        else
        {
            temp = head;
            while(temp->next != head)
                temp = temp->next;
            ptr->next = head;
            temp -> next = ptr;
            head = ptr;
        }
        printf("\nnode inserted\n");
    }

}
void lastinsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node *)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW\n");
    }
    else
    {
        printf("\nEnter Data?");
        scanf("%d",&item);
```

```
        ptr->data = item;
        if(head == NULL)
        {
            head = ptr;
            ptr -> next = head;
        }
        else
        {
            temp = head;
            while(temp -> next != head)
            {
                temp = temp -> next;
            }
            temp -> next = ptr;
            ptr -> next = head;
        }

        printf("\nnode inserted\n");
    }

}

void display()
{
    struct node *ptr;
    ptr=head;
    if(head == NULL)
    {
        printf("\nnothing to print");
    }
    else
    {
        printf("\n printing values ... \n");
```

```
        while(ptr -> next != head)
        {

            printf("%d\n", ptr -> data);
            ptr = ptr -> next;
        }
        printf("%d\n", ptr -> data);
    }

}
```

Output:

```
***Main Menu***

Choose one option from the following list ...

================================================

1.Insert in begining
2.Insert at last
3.Display
4.Exit

Enter your choice?
1

Enter the node data?
10

node inserted
```

Experiment No.: 5

Program Description:

Implementation of Circular Doubly Linked List using Pointers.

Solution:

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

//function declaration

void createDCLL();
void dispay();

struct node{
        int data;
        struct node *next;
        struct node *prev;
    }*head, *tail;

int main(){

createDCLL();
dispay();

return 0;
}
void createDCLL(){
    struct node *newnode;
    int choice=1;
    head=0;
    while(choice){
        newnode=(struct node *)malloc(sizeof(struct node));
        printf("Enter data: ");
        scanf("%d",&newnode->data);
        if(head==0){
            head=tail=newnode;
            head->next=head;
            head->prev=head;
        }
        else{
            tail->next=newnode;
            newnode->prev=tail;
            newnode->next=head;
            head->prev=newnode;
```

```
            tail=newnode;
        }
        printf("Do you want to continue (Press 1 for YES and 0 for
NO)?? ");
        scanf("%d",&choice);
    }
}



void dispay(){
    struct node *temp;
    temp=head;
    if(head==0){
        printf("List is empty!");
    }
    else{
        while(temp!=tail){
            printf("%d ", temp->data);
            temp=temp->next;
        }
        printf("%d",temp->data);
    }
}
```

Output:

```
Enter data: 10

Do you want to continue (Press 1 for YES and 0 for NO)?? 1

Enter data: 20

Do you want to continue (Press 1 for YES and 0 for NO)?? 1

Enter data: 30

Do you want to continue (Press 1 for YES and 0 for NO)?? 0

10 20 30
```

Section-B (Stack)

Experiment No.: 1

Program Description:

Implementation of Stack using Array.

Solution:

```
#include <stdio.h>

#define SIZE 5

// gobal variable declaration of the array index and array.
 int top = -1;
 int arr[SIZE];

// function prototyping.
void push(int val);
void pop();
void display();

int main()
{
  push(5);
  push(9);
  push(3);
  pop();
  display();

  return 0;
}
// function definitions
 void push(int val)
 {
    if (top == SIZE)
   {
    printf("stack is full\n");
   }
  else
  {
    top++;
    arr[top] = val;
    printf("element %d is successfully inserted\n", val);
```

```
 }
 }
void display()
 {
  printf("\nelements of stack\n");
  for (int top = 0; top < SIZE; top++)
  {
    if (arr[top] == 0)
    {
     printf("\ntotal empty space left in stack is: %d\n\n",SIZE-(top));
      break;
    }
    else {
      printf("%d\n", arr[top]);
    }
  }
void pop()
{
  if (top == -1)
  {
    printf("stack is empty\n");
  }
  else
  {
    printf("element %d is deleted from stack\n");
    arr[top] = 0;
    top--;
  }
}
```

Output:

```
Element 5 is successfully inserted

Element 9 is successfully inserted

Element 3 is successfully inserted

Element 3 is deleted from stack

Elements of stack:

5

9

Total empty space left in stack: 3
```

Experiment No.: 2

Program Description:

Implementation of Stack using Pointers.

Solution:

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

// Decleration of structure
struct node
{
  /* elements of node */
  int data;
  struct node *next;
};
struct node *Head = NULL;

// function prototyping
void push(int val);
void pop();
void display_stack();

int main(void)
{
  push(10);
  push(20);
  push(30);
  pop();
  display_stack();
}

void push(int val)
{
  struct node *Newnode = malloc(sizeof(struct node));
  Newnode->data = val;

  Newnode->next = Head;

  Head = Newnode;
  printf("element %d is successfully inserted\n", Head->data);
```

```
}

void pop()
{
  struct node *temp;

  if (Head == NULL)
    printf("stack is empty\n");

  else
  {
    printf("Poped element = %d\n", Head->data);
    temp = Head;
    Head = Head->next;
    free(temp);
  }
}

void display_stack()
{
  struct node *current = Head;
  printf("top ->");
  while (current != NULL)
  {
    printf(" %d ->", current->data);
    current = current->next;
  }
  printf(" Bottom");
}
```

Output:

```
element 10 is successfully inserted

element 20 is successfully inserted

element 30 is successfully inserted

Poped element = 30

top -> 20 -> 10 -> Bottom
```

Experiment No.: 3

Program Description:

Program for Tower of Hanoi using recursion.

Solution:

```
#include <stdio.h>

// C recursive function to solve tower of hanoi puzzle
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
  if (n == 1)
  {
    printf("\n Move disk 1 from rod %c to rod %c", from_rod,
to_rod);
    return;
  }
  towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);
  printf("\n Move disk %d from rod %c to rod %c", n, from_rod,
to_rod);
  towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);
}

int main()
{
  int n = 4;                        // Number of disks
  towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
  return 0;
}
```

Output:

```
Move disk 1 from rod A to rod C

Move disk 2 from rod A to rod B

Move disk 1 from rod C to rod B

Move disk 3 from rod A to rod C

Move disk 1 from rod B to rod A

Move disk 2 from rod B to rod C

Move disk 1 from rod A to rod C
```

```
Move disk 4 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 2 from rod C to rod A
Move disk 1 from rod B to rod A
Move disk 3 from rod C to rod B
Move disk 1 from rod A to rod C
Move disk 2 from rod A to rod B
Move disk 1 from rod C to rod B
```

Experiment No.: 4

Program Description:

Program to find out factorial of given number using recursion. Also show the various states of stack using in this program.

Solution:

```
#include <stdio.h>
// function prototyping
int factorial(int N);

int main()
{
  int n, ans;
  printf("Enter a number\n");
  scanf("%d", &n);

  if (n >= 0)
  {
    ans = factorial(n);
    printf("%d\n", ans);

    return 0;
  }
}
// function definition
int factorial(int N)
{
  if (N <= 1)
    return 1;

  return N * factorial(N - 1);
}
```

Output:

```
Enter a number

5

120
```

Section-C (Queue)

Experiment No.: 1

Program Description:

Implementation of Queue using Array.

Solution:

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

// global variable declaration for Queue
int arr[size];
int Front = 0;
int rear = 0;

// function prototyping
void enqueue(int val);
void dequeue();
void display_Queue();

int main()
{
  enqueue(10);
  enqueue(20);
  enqueue(30);
  enqueue(40);
  dequeue();
  dequeue();
  enqueue(50);
  display_Queue();
  return 0;
}

void enqueue(int val)
{
  if (rear == size)
    printf("Queue is full\n");

  else
  {
    arr[rear] = val;
```

```
    printf("element %d is successfully inserted\n", arr[rear]);
    rear++;
  }
}

void dequeue()
{
  if (Front == rear)
    printf("Queue is empty\n");

  else
  {
    printf("Dequeued element= %d\n", arr[Front]);
    Front++;
  }
}

void display_Queue()
{
  int i = 0;
  for (i = Front; i < size; i++)
  {
    printf("%d ", arr[Front]);
    Front++;
  }
}
```

Output:

```
element 10 is successfully inserted

element 20 is successfully inserted

element 30 is successfully inserted

element 40 is successfully inserted

Dequeued element= 10

Dequeued element= 20

element 50 is successfully inserted

Elements in the Queue: 30 40 50
```

Experiment No.: 2

Program Description:

Implementation of Queue using pointer.

Solution:

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

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

struct node *Front = NULL, *rear = NULL;

// function prototyping
void enqueue(int val);
void dequeue();
void display_Queue();

int main()
{
  enqueue(10);
  enqueue(20);
  enqueue(30);
  enqueue(40);
  enqueue(50);
  dequeue();
  dequeue();
  dequeue();
  dequeue();
  enqueue(60);
  enqueue(70);
  display_Queue();
  return 0;
}
```

```
// function definition
void enqueue(int val)
{
  struct node *Newnode = malloc(sizeof(struct node));
  Newnode->data = val;
  Newnode->next = NULL;

  if (Front == NULL && rear == NULL)
    Front = rear = Newnode;

  else
  {
    rear->next = Newnode;
    rear = Newnode;
  }
  printf("element %d is successfully inserted\n", rear->data);
}

void dequeue()
{
  struct node *temp;
  if (Front == NULL)
    printf("Queue is empty.Unable to perform dequeue\n");
  else
  {
    printf("Dequeued element = %d\n", Front->data);

    temp = Front;
    Front = Front->next;

    if (Front == NULL)
      rear = NULL;

    free(temp);
  }
}

void display_Queue()
{
  struct node *current;
  current = Front;
  printf("Front -> ");
```

```
  while (current != NULL)
  {
    printf("%d ->", current->data);
    current = current->next;
  }
  printf(" rear");
}
```

Output:

```
element 10 is successfully inserted
element 20 is successfully inserted
element 30 is successfully inserted
element 40 is successfully inserted
element 50 is successfully inserted
Dequeued element = 10
Dequeued element = 20
Dequeued element = 30
Dequeued element = 40
element 60 is successfully inserted
element 70 is successfully inserted
Front -> 50 -> 60 -> 70 -> rear
```

Experiment No.: 3

Program Description:

Implementation of Circular Queue using Array.

Solution:

```
#include <stdio.h>
#include <stdbool.h>
#define size 5

// Global variable declaration
int arr[size];
int Front = -1;
int rear = -1;

// function prototyping
void enqueue(int val);
int dequeue();
void display_Queue();
bool isQueueFull();

int main()
{
  enqueue(10);
  enqueue(20);
  enqueue(30);
  enqueue(40);
  dequeue();
  dequeue();
  dequeue();
  enqueue(60);
  enqueue(70);
  display_Queue();
  return 0;
}
void enqueue(int val)
{
  if (isQueueFull())
  {
    printf("Circular Queue is Full.Unable to insert element\n");
  }
```

```
  else
  {
    if (Front == -1)
      Front = 0;

    rear = (rear + 1) % size;
    arr[rear] = val;

    printf("element %d is successfully inserted\n", arr[rear]);
  }
}

int dequeue()
{
  if (isQueueFull())
  {
    printf("Circular Queue is empty\n");
    return -1;
  }
  else
  {
    int val = arr[Front];

    if (Front == rear)
    {
      Front = -1;
      rear = -1;
    }
    else
      Front = (Front + 1) % size;

    return val;
  }
}

void display_Queue()
{
  int i;
  if (isQueueFull())
    printf("Queue is empty\n");
 else{
```

```
    printf("\n Front -> %d", Front);
    printf("\n Items -> ");
    for (i = Front; i != rear; i = (i + 1) % size)
    {
      printf("%d ", arr[i]);
    }
    printf("%d ", arr[i]);
    printf("\n Rear -> %d\n", rear);
  }
}

bool isQueueFull()
{
  if (Front == 0 && rear == size - 1)
    return true;

  if (rear == Front - 1)
    return true;

  return false;
}
```

Output:

```
element 10 is successfully inserted

element 20 is successfully inserted

element 30 is successfully inserted

element 40 is successfully inserted

Dequeued element = 10

Dequeued element = 20

Dequeued element = 30

element 60 is successfully inserted

element 70 is successfully inserted


 Front -> 0

 Items -> 40 60 70

 Rear ->
```

Section-D (Trees)

Experiment No.: 1

Program Description:

Implementation of Binary Search Tree.

Solution:

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

struct node {
  int data;
  struct node *right_child;
  struct node *left_child;
};
  struct node* new_node(int x){
  struct node *temp;
  temp = malloc(sizeof(struct node));
  temp->data = x;
  temp->left_child = NULL;
  temp->right_child = NULL;
    return temp;
}
struct node* search(struct node * root, int x){
  if (root == NULL || root->data == x)
    return root;
  else if (x > root->data)
    return search(root->right_child, x);
  else
    return search(root->left_child, x);
}
```

```
struct node* insert(struct node * root, int x){
  if (root == NULL)
    return new_node(x);
  else if (x > root->data)
    root->right_child = insert(root->right_child, x);
  else
    root -> left_child = insert(root->left_child, x);
  return root;
}
struct node* find_minimum(struct node * root) {
  if (root == NULL)
     return NULL;
  else if (root->left_child != NULL)
    return find_minimum(root->left_child);
  return root;
}
  struct node* delete(struct node * root, int x)
{
  if (root == NULL)
    return NULL;
  if (x > root->data)
    root->right_child = delete(root->right_child, x);
  else if (x < root->data)
    root->left_child = delete(root->left_child, x);
  else {
    if (root->left_child == NULL && root->right_child == NULL){
      free(root);
      return NULL;
    }
    else if (root->left_child == NULL || root->right_child == NULL){
```

```
      struct node *temp;
      if (root->left_child == NULL)
        temp = root->right_child;
      else
        temp = root->left_child;
      free(root);
      return temp;
    }
    else {
      struct node *temp = find_minimum(root->right_child);
      root->data = temp->data;
      root->right_child = delete(root->right_child, temp->data);
    }
  }
  return root;
}
void inorder(struct node *root)
{
  if (root != NULL)
  {
    inorder(root->left_child);
    printf(" %d ", root->data);
    inorder(root->right_child);
  }
}
  int main() {
  struct node *root;
  root = new_node(20);
  insert(root, 5);
  insert(root, 1);
```

```
  insert(root, 15);
  insert(root, 9);
  insert(root, 7);
  insert(root, 12);
  insert(root, 30);
  insert(root, 25);
  insert(root, 40);
  insert(root, 45);
  insert(root, 42);

  inorder(root);
  printf("\n");
  root = delete(root, 15);
  root = delete(root, 40);
  root = delete(root, 9);

  inorder(root);
  printf("\n");
  return 0;
}
```

Output:

```
1  5  7  9  12  15  20  25  30  35  40  42  45
1  5  7  12  15  20  25  30  35  42  45
```

Experiment No.: 2

Program Description:

Conversion of BST PreOrder/PostOrder/InOrder.

Solution:

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

typedef struct Node {
    int data;
    struct Node* left;
    struct Node* right;
} Node;

Node* newNode(int data) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

void inorderTraversal(Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}

void preorderTraversal(Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorderTraversal(root->left);
        preorderTraversal(root->right);
    }
}

void postorderTraversal(Node* root) {
    if (root != NULL) {
        postorderTraversal(root->left);
        postorderTraversal(root->right);
        printf("%d ", root->data);
    }
}

Node* insert(Node* root, int data) {
```

```
    if (root == NULL) {
        return newNode(data);
    }

    if (data < root->data) {
        root->left = insert(root->left, data);
    } else if (data > root->data) {
        root->right = insert(root->right, data);
    }

    return root;
}

int main() {
    Node* root = NULL;

    // Inserting elements to create a BST
    int elements[] = {8, 3, 10, 1, 6, 9, 14};
    int numElements = sizeof(elements) / sizeof(elements[0]);

    for (int i = 0; i < numElements; ++i) {
        root = insert(root, elements[i]);
    }

    printf("Inorder Traversal: ");
    inorderTraversal(root);
    printf("\n");

    printf("Preorder Traversal: ");
    preorderTraversal(root);
    printf("\n");

    printf("Postorder Traversal: ");
    postorderTraversal(root);
    printf("\n");

    return 0;
}
```

Output:

```
Inorder Traversal: 1 3 6 8 9 10 14

Preorder Traversal: 8 3 1 6 10 9 14

Postorder Traversal: 1 6 3 9 14 10 8
```

Experiment No.: 3

Program Description:

Implementation of Kruskal Algorithm

Solution:

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

// Kruskal's algorithm
struct Edge
{
  int src, dest, weight;
};
struct Subset
{
  int parent;
  int rank;
};

int compareEdges(const void *a, const void *b);
int find(struct Subset subsets[], int i);
void unionSets(struct Subset subsets[], int x, int y);
void kruskalMST(struct Edge edges[], int V, int E);

int main()
{
  int V = 4;
  int E = 5;

  struct Edge edges[] = {
      {0, 1, 10},
      {0, 2, 6},
      {0, 3, 5},
      {1, 3, 15},
      {2, 3, 4}};

  kruskalMST(edges, V, E);

  return 0;
}
```

```
int compareEdges(const void *a, const void *b)
{
  return ((struct Edge *)a)->weight - ((struct Edge *)b)->weight;
}

int find(struct Subset subsets[], int i)
{
  if (subsets[i].parent != i)
    subsets[i].parent = find(subsets, subsets[i].parent);
  return subsets[i].parent;
}

void unionSets(struct Subset subsets[], int x, int y)
{
  int rootX = find(subsets, x);
  int rootY = find(subsets, y);

  if (subsets[rootX].rank < subsets[rootY].rank)
    subsets[rootX].parent = rootY;
  else if (subsets[rootX].rank > subsets[rootY].rank)
    subsets[rootY].parent = rootX;
  else
  {
    subsets[rootY].parent = rootX;
    subsets[rootX].rank++;
  }
}
void kruskalMST(struct Edge edges[], int V, int E)
{
  struct Subset *subsets = (struct Subset *)malloc(V * sizeof(struct
Subset));

  for (int v = 0; v < V; v++)
  {
    subsets[v].parent = v;
    subsets[v].rank = 0;
  }
  qsort(edges, E, sizeof(edges[0]), compareEdges);

  printf("Minimum Spanning Tree:\n");
```

```
  int i = 0, e = 0;
  while (e < V - 1 && i < E)
  {
    struct Edge nextEdge = edges[i++];
    int x = find(subsets, nextEdge.src);
    int y = find(subsets, nextEdge.dest);

    if (x != y)
    {
      printf("%d -- %d == %d\n", nextEdge.src, nextEdge.dest,
nextEdge.weight);
      unionSets(subsets, x, y);
      e++;
    }
  }
  free(subsets);
}
```

Output:

```
Minimum Spanning Tree:
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
```

Experiment No.: 4

Program Description:

Implementation of Prim Algorithm

Solution:

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

// Number of vertices in the graph
#define V 5

int minKey(int key[], bool mstSet[]) {
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++) {
        if (mstSet[v] == false && key[v] < min) {
            min = key[v];
            min_index = v;
        }
    }

    return min_index;
}

void printMST(int parent[], int graph[V][V]) {
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d \t%d \n", parent[i], i,
graph[i][parent[i]]);
}

void primMST(int graph[V][V]) {
    int parent[V]; // To store constructed MST
    int key[V];    // Key values used to pick minimum weight edge in
cut
    bool mstSet[V]; // To represent set of vertices included in MST

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++) {
        key[i] = INT_MAX;
        mstSet[i] = false;
    }

    // Always include the first vertex in MST.
```

```
    key[0] = 0; //Make key 0 so that this vertex is picked as the
first vertex
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum key vertex from the set of vertices not
yet included in MST

        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of the adjacent
vertices of the picked vertex
        for (int v = 0; v < V; v++) {
            if (graph[u][v] && mstSet[v] == false && graph[u][v] <
key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }

    // Print the constructed MST

    printMST(parent, graph);
}

int main() {
    // Example graph represented by its adjacency matrix
    int graph[V][V] = {
        {0, 2, 0, 6, 0},
        {2, 0, 3, 8, 5},
        {0, 3, 0, 0, 7},
        {6, 8, 0, 0, 9},
        {0, 5, 7, 9, 0}
    };

    // Print the MST using Prim's algorithm
    primMST(graph);

    return 0;
}
```

Output:

```
Edge    Weight
0 - 1    2
1 - 2    3
0 - 3    6
2 - 4    7
```

Experiment No.: 5

Program Description:

Implementation of Dijkstra Algorithm

Solution:

```
#include <stdio.h>
#include <limits.h>
#include<stdbool.h>

#define V 6   // Number of vertices

int minDistance(int dist[], bool sptSet[]) {
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++) {
        if (!sptSet[v] && dist[v] <= min) {
            min = dist[v];
            min_index = v;
        }
    }

    return min_index;
}

void printSolution(int dist[]) {
    printf("Vertex \t Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d \t\t %d\n", i, dist[i]);
}

void dijkstra(int graph[V][V], int src) {
    int dist[V];     // The output array dist[i] holds the shortest
distance from src to i
    bool sptSet[V];  // sptSet[i] will be true if vertex i is
included in the shortest path tree or the shortest distance from src
to i is finalized

    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX;
        sptSet[i] = false;
    }

    dist[src] = 0;

    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);
```

```
        sptSet[u] = true;

        for (int v = 0; v < V; v++)
           if  (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX &&
dist[u] +            graph[u][v] < dist[v])

                dist[v] = dist[u] + graph[u][v];
    }

    printSolution(dist);
}

int main() {
    int graph[V][V] = {
        {0, 1, 4, 0, 0, 0},
        {0, 0, 2, 7, 0, 0},
        {0, 0, 0, 0, 3, 0},
        {0, 0, 0, 0, 0, 1},
        {0, 0, 0, 2, 0, 5},
        {0, 0, 0, 0, 0, 0}
    };

    dijkstra(graph, 0);

    return 0;
}
```

Output:

```
Vertex    Distance from Source
0         0
1         1
2         3
3         8
4         6
5         9
```

Section-E (Sorting & Searching)

Experiment No.: 1

Program Description:

Implementation of Sorting

Solution:

Merge Sort

```
#include<stdio.h>

// function prototyping
void MergeSort(int arr[],int start,int end);
void Merge(int arr[],int start,int mid,int end);

int main()
{
 int arr[]={2,5,8,3,9,4,21,10,12};
 int size=sizeof(arr)/sizeof(arr[0]);
 int start=0;
 int end=size-1;

  MergeSort(arr,start,end);

  // print the array
  for(int i=start;i<=end;i++)
   {
    printf("%d ",arr[i]);
   }
}

// function defination
void MergeSort(int arr[],int start,int end)
{
  if (start<end)
  {
   int mid=(start+end)/2;
    MergeSort(arr,start,mid);
    MergeSort(arr,mid+1,end);
    Merge(arr,start,mid,end);
```

```
  }


}

void Merge(int arr[],int start,int mid,int end)

 int temp[end-start+1];

  int i,j,k;
  i=start;
  j=mid+1;
  k=0;

  while (i<=mid && j<=end)
  {
    if(arr[i]<arr[j])
    {
      temp[k]=arr[i];
      i++;
      k++;
    }
    else
    {
       temp[k]=arr[j];
       j++;
       k++;
    }
  }

  while (i<=mid)
  {
    temp[k]=arr[i];
    i++;
    k++;
  }

  while (j<=end)
  {
    temp[k]=arr[j];
    j++;
    k++;
```

```
  }

 k=0;
 for(i=start;i<=end;i++)
 arr[i]=temp[k++];
}
```

Output:

```
 2 3 4 5 8 9 10 12 21
```

Bubble Sort

```
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void bubbleSort(int arr[], int n) {
    for (int I = 0; I < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                swap(&arr[j], &arr[j+1]);
            }
        }
    }
}

void printArray(int arr[], int size) {
    for (int I = 0; I < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Unsorted array: \n");
    printArray(arr, n);

    bubbleSort(arr, n);

    printf("Sorted array: \n");
    printArray(arr, n);

    return 0;
}
```

Output:

```
Unsorted array:
64 34 25 12 22 11 90
Sorted array:
11 12 22 25 34 64 90
```

Insertion Sort

```
#include <stdio.h>

void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1] that are greater than key to
one position ahead of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Unsorted array: \n");
    printArray(arr, n);

    insertionSort(arr, n);

    printf("Sorted array: \n");
    printArray(arr, n);

    return 0;
}
```

Output:

```
Unsorted array:
12 11 13 5 6
Sorted array:
5 6 11 12 13
```

Selection Sort

```
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *a = temp;
}

void selectionSort(int arr[], int n) {
    int i, j, min_idx;

    // One by one move the boundary of the unsorted subarray
    for (i = 0; i < n-1; i++) {
        // Find the minimum element in the unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Unsorted array: \n");
    printArray(arr, n);

    selectionSort(arr, n);

    printf("Sorted array: \n");
```

```
    printArray(arr, n);

    return 0;
}
```

Output:

```
Unsorted array:
64 25 12 22 11
Sorted array:
11 12 22 25 64
```

Quick sort

```
#include <stdio.h>

void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Choose the pivot as the last element
    int i = (low - 1); // Index of smaller element

    for (int j = low; j <= high - 1; j++) {
        // If the current element is smaller than or equal to the
pivot
        if (arr[j] <= pivot) {
            i++; // Increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Partitioning index
        int pi = partition(arr, low, high);

        // Recursively sort elements before and after the partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
```

```
}

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Unsorted array: \n");
    printArray(arr, n);

    quickSort(arr, 0, n - 1);

    printf("Sorted array: \n");
    printArray(arr, n);

    return 0;
}
```

Output:

```
Unsorted array:
10 7 8 9 1 5
Sorted array:
1 5 7 8 9 10
```

Experiment No.: 2

Program Description:

Implementation of Binary Search on a list of numbers stored in an Array

Solution:

```
#include <stdio.h>

int binary_search(int arr[], int size, int target)
{
  // function
  int low = 0;
  int high = size - 1;

  while (low <= high)
  {
    int mid = (low + high) / 2;
    int mid_value = arr[mid];

    if (mid_value == target)
    {
      return mid; // Target found, return the index
    }
    else if (mid_value < target)
    {
      low = mid + 1; // Search the right half
    }
    else
    {
      high = mid - 1; // Search the left half
    }
  }

  return -1; // Target not found
}

int main()
{
  // main function
  int sorted_array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int size = sizeof(sorted_array) / sizeof(sorted_array[0]);
```

```
  int target_value = 1;
  // set target value.

  int result = binary_search(sorted_array, size, target_value);
  // function call.

  if (result != -1)
  {
    printf("Target %d found at index %d\n", target_value, result);
  }
  else
  {
    printf("Target %d not found in the array\n", target_value);
  }

  return 0;
}
```

Output:

```
Target 1 found at index 0
```

Experiment No.: 3

Program Description:

Implementation of Binary Search on a list of strings stored in an Array

Solution:

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

int string_binary_search(char arr[][50], int size, char target[])
{ // function
  int low = 0;
  int high = size - 1;

  while (low <= high)
  {
    int mid = (low + high) / 2;
    int compare_result = strcmp(arr[mid], target);

    if (compare_result == 0)
    {
      return mid; // Target found, return the index
    }
    else if (compare_result < 0)
    {
      low = mid + 1; // Search the right half
    }
    else
    {
      high = mid - 1; // Search the left half
    }
  }

  return -1; // Target not found
}

int main()
{ // Main function.
  char sorted_strings[][50] = {"apple", "banana", "cherry", "date",
"grape", "kiwi", "orange", "pear"};
  int size = sizeof(sorted_strings) / sizeof(sorted_strings[0]);
```

```
  char target_string[] = "apple"; // set target string

  int result = string_binary_search(sorted_strings, size,
target_string);

  if (result != -1)
  {
    printf("Target %s found at index %d\n", target_string, result);
  }
  else
  {
    printf("Target %s not found in the array\n", target_string);
  }

  return 0;
}
```

Output:

```
Target apple found at index 0
```

Experiment No.: 4

Program Description:

Implementation of Linear Search on a list of strings stored in an Array

Solution:

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

int linear_search(char arr[][50], int size, char target[]) {
    //linear search function.
    for (int i = 0; i < size; i++) {
        if (strcmp(arr[i], target) == 0) {
            return i;  // Target found, return the index
        }
    }

    return -1;  // Target not found
}

int main()
{
    //Main function.
    char strings[][50] = {"apple", "banana", "cherry", "date",
"grape", "kiwi", "orange", "pear"};
    int size = sizeof(strings) / sizeof(strings[0]);
    char target_string[] = "guava";
    //set target string value.

    int result = linear_search(strings, size, target_string);

    if (result != -1) {
        printf("Target %s found at index %d\n", target_string,
result);
    } else {
        printf("Target %s not found in the array\n", target_string);
    }

    return 0;
}
```

Output:

```
Target guava not found in the array
```

Experiment No.: 5

Program Description:

Implementation of Binary Search on a list of strings stored in a Single Linked List

Solution:

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

// Define a node structure for the linked list
struct Node {
    char data[50];
    struct Node* next;
};

// Function to insert a new node at the end of the linked list
void insert(struct Node** head, char data[]) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
    strcpy(newNode->data, data);
    newNode->next = NULL;

    if (*head == NULL) {
        *head = newNode;
        return;
    }

    struct Node* current = *head;
    while (current->next != NULL) {
        current = current->next;
    }

    current->next = newNode;
}

// Function to perform binary search on a linked list
int linked_list_binary_search(struct Node* head, char target[]) {
    int index = 0;
    struct Node* current = head;

    while (current != NULL && strcmp(current->data, target) < 0) {
        current = current->next;
        index++;
    }

    if (current != NULL && strcmp(current->data, target) == 0) {
```

```
        return index;  // Target found, return the index
    }

    return -1;  // Target not found
}

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

    // Insert elements into the linked list
    insert(&head, "apple");
    insert(&head, "banana");
    insert(&head, "cherry");
    insert(&head, "date");
    insert(&head, "grape");
    insert(&head, "kiwi");
    insert(&head, "orange");
    insert(&head, "pear");

    char target_string[] = "strawberry";

    int result = linked_list_binary_search(head, target_string);

    if (result != -1) {
        printf("Target %s found at index %d\n", target_string,
result);
    } else {
        printf("Target %s not found in the linked list\n",
target_string);
    }

    // Free the memory allocated for the linked list
    struct Node* current = head;
    while (current != NULL) {
        struct Node* next = current->next;
        free(current);
        current = next;
    }

    return 0;
}
```

Output:

```
Target strawberry not found in the linked list

Target banana found at index 1
```