**Program for Heap Sort**

**ALGORITHM**

**conversion(string a[])**

**BEGIN:**

1. Build a max heap from the input array. This is done by starting at the first non-leaf node and performing the **heapify()** operation on each node in reverse level order. This ensures that the max heap property is satisfied at each node in the heap.
2. Once the max heap is built, the maximum element is at the root of the heap (i.e., the first element in the array). Swap this element with the last element in the heap, which is the element in the last position of the array.
3. Since the maximum element is now at the end of the array, we no longer need to consider it in the heap. Reduce the size of the heap by one and perform **heapify()** on the root of the heap to restore the max heap property.
4. Repeat steps 2 and 3 until the heap has only one element left (i.e., the array is fully sorted).

Top of Form

**END;**

**Implementation**

#include <stdio.h>

void heapify(int arr[], int n, int i) {

int largest = i;

int l = 2 \* i + 1;

int r = 2 \* i + 2;

if (l < n && arr[l] > arr[largest])

largest = l;

if (r < n && arr[r] > arr[largest])

largest = r;

if (largest != i) {

int temp = arr[i];

arr[i] = arr[largest];

arr[largest] = temp;

heapify(arr, n, largest);

}}

void heapSort(int arr[], int n) {

for (int i = n / 2 - 1; i >= 0; i--)

heapify(arr, n, i);

for (int i = n - 1; i >= 0; i--) {

int temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

heapify(arr, i, 0);

}}

void printArray(int arr[], int n) {

for (int i = 0; i < n; i++)

printf("%d ", arr[i]);

printf("\n");

}

int main() {

int arr[] = {12, 11, 13, 5, 6, 7};

int n = sizeof(arr) / sizeof(arr[0]);

printf("Original array: \n");

printArray(arr, n);

heapSort(arr, n);

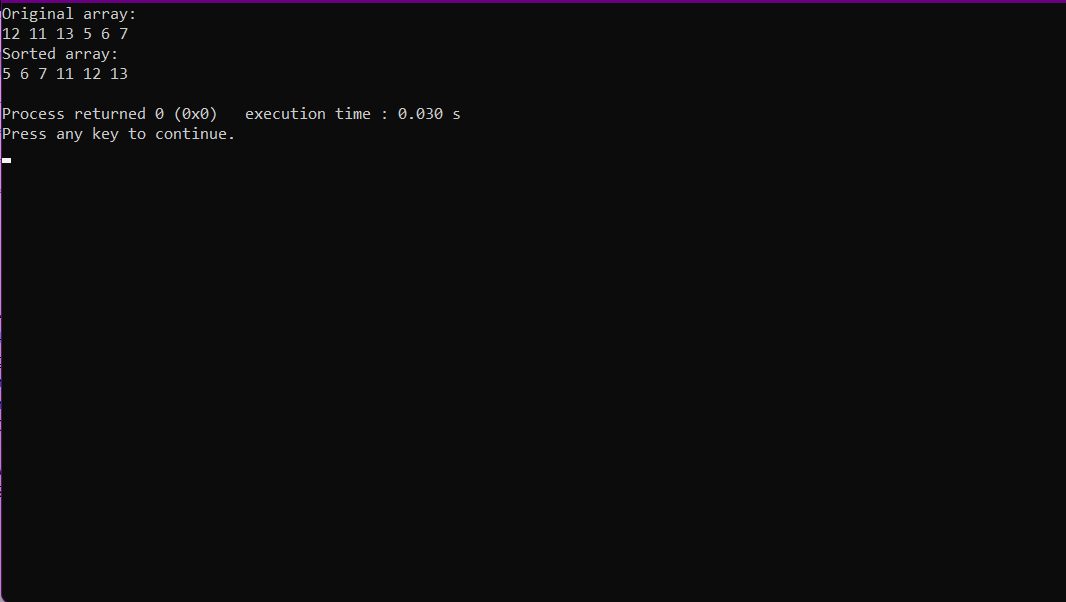
printf("Sorted array: \n");

printArray(arr, n);

return 0;

}

**RESULT:**



**Practical-4: Program for implementation of stacks using a single array.**

**Array Algorithm:**

• We will start two stacks from two extreme end of input array. Both these stacks will grow towards each other.

• Left stack will start index 0 and grow towards right end of array.

• Right array will start from index N-1 and grow towards left end of array.

• We will use stack number to differentiate between these two arrays. 0 and 1 will be used for left and right array respectively.

• When both stacks meet each other then we won't be able to push any element in any stack.

• Here is the prototype of push and pop operations.

**PUSH**

• void push(int stack, int num);

• It will take stack number and integer to be pushed in stack as input.

**POP**

• int pop(int stack)

• It takes stack number as input. It removes the top element from stack corresponding to passed stack number.

**Implementation**

#include <stdio.h>

#define SIZE 10

int ar[SIZE];

int top1 = -1;

int top2 = SIZE;

//Functions to push data

void push\_stack1 (int data)

{if (top1 < top2 - 1)

{ar[++top1] = data;}

else

{printf ("Stack Full! Cannot Push\n");

}}

void push\_stack2 (int data)

{if (top1 < top2 - 1)

{ar[--top2] = data;}

else

{printf ("Stack Full! Cannot Push\n");

}}

//Functions to pop data

void pop\_stack1 ()

{if (top1 >= 0)

{int popped\_value = ar[top1--];

printf ("%d is being popped from Stack 1\n", popped\_value); }

else

{printf ("Stack Empty! Cannot Pop\n");}}

void pop\_stack2 ()

{if (top2 < SIZE)

{int popped\_value = ar[top2++];

printf ("%d is being popped from Stack 2\n", popped\_value); }

else

{printf ("Stack Empty! Cannot Pop\n");}}

//Functions to Print Stack 1 and Stack 2

void print\_stack1 ()

{int i;

for (i = top1; i >= 0; --i)

{printf ("%d ", ar[i]);}

printf ("\n");}

void print\_stack2 ()

{int i;

for (i = top2; i < SIZE; ++i)

{printf ("%d ", ar[i]);}

printf ("\n");}

int main()

{int ar[SIZE];

int i;

int num\_of\_ele;

printf("ARYAN CHOUDHARYRoll.No: 2100320120034\n CS-A\n");

printf ("We can push a total of 10 values\n");

//Number of elements pushed in stack 1 is 6

//Number of elements pushed in stack 2 is 4

for (i = 1; i <= 6; ++i)

{push\_stack1 (i);

printf ("Value Pushed in Stack 1 is %d\n", i);}

for (i = 1; i <= 4; ++i)

{push\_stack2 (i);

printf ("Value Pushed in Stack 2 is %d\n", i);}

print\_stack1 ();

print\_stack2 ();

printf ("Pushing Value in Stack 1 is %d\n", 11);

push\_stack1 (11);

num\_of\_ele = top1 + 1;

while (num\_of\_ele)

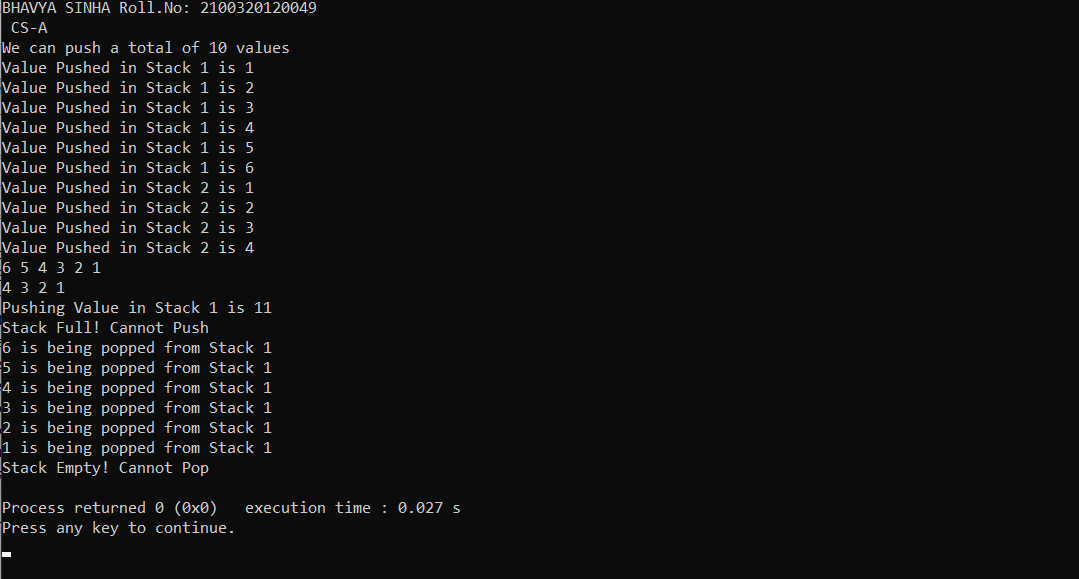
{pop\_stack1 ();

--num\_of\_ele;}

pop\_stack1 ();

return 0;}

**RESULT**



**Practical-5: Program of Array Implementation of Linear Queue**

⦁**ALGORITHM INTIALIZE(QUEUE Q)**

BEGIN:

Q.REAR=0

Q.FRONT=1

END;

⦁**ALGORITHM ENQUEUE(QUEUE Q,key)**

BEGIN:

IF Q.REAR ==SIZE THEN

write (Queue overflow)

Exit 1

Q.REAR=Q.REAR+1

Q.item[Q.REAR]=key

END;

⦁ **ALGORITHM DEQUEUE(QUEUE Q)**

BEGIN:

IF Q.REAR-Q.FRONT+1==0 THEN

Exit(1)

x=Q.item[Q.FRONT]

Q.FRONT =Q.FRONT +1

RETURN x

END;

⦁**ALGORITHM EMPTY(QUEUE Q)**

BEGIN:

IF Q.REAR – Q.FRONT +1 == 0 THEN

RETURN TRUE

ELSE

RETURN FALSE

END;

For all the above ALGORITHM

**Time Complexity: ϴ (1)**

**Space Complexity: ϴ (1)**

**Implementation**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

typedef struct {

int data[MAX\_SIZE];

int front;

int rear;} Queue;

void enqueue(Queue \*q, int x) {

if (q->rear == MAX\_SIZE - 1) {

printf("Queue overflow\n");

exit(1);} q->data[++(q->rear)] = x;}

int dequeue(Queue \*q) {

if (q->front == q->rear) {

printf("Queue underflow\n");

exit(1); }

return q->data[++(q->front)];}

int is\_empty(Queue \*q) {

return q->front == q->rear;}

int is\_full(Queue \*q) {

return q->rear == MAX\_SIZE - 1;}

void display(Queue \*q) {if (is\_empty(q)) {

printf("Queue is empty\n");

return;}

for (int i = q->front + 1; i <= q->rear; i++) {

printf("%d ", q->data[i]);}printf("\n");}

int main() {

Queue q;

q.front = q.rear = -1;

int choice, x;

while (1) {

printf("1. Enqueue\n");

printf("2. Dequeue\n");

printf("3. Display\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the element to be enqueued: ");

scanf("%d", &x);

if (is\_full(&q)) {

printf("Queue is full\n");

} else{ enqueue(&q, x);}

break;

case 2:

if (is\_empty(&q)) {printf("Queue is empty\n"); } else {printf("Dequeued element is %d\n", dequeue(&q));}

break;

case 3:

display(&q);

break;

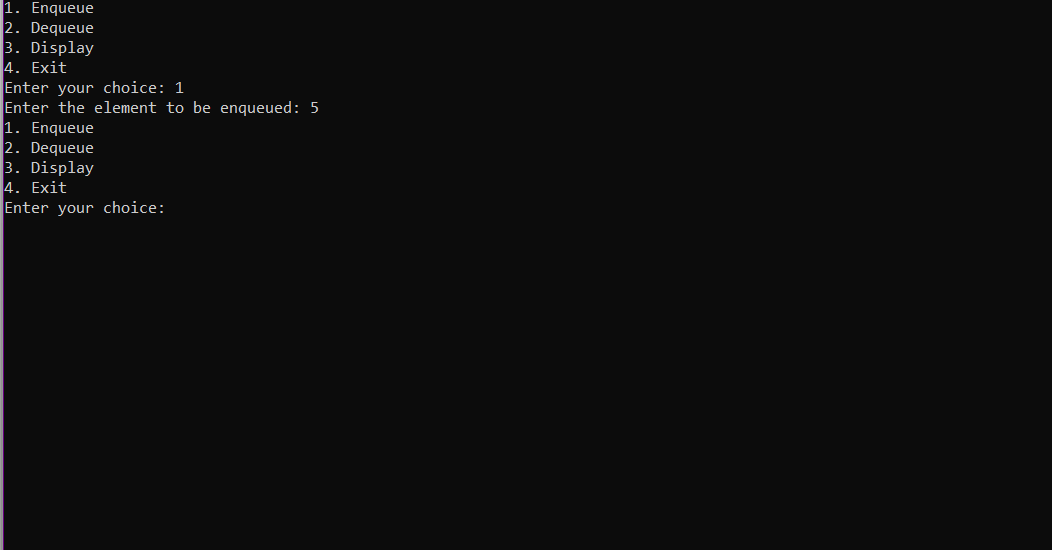
case 4:

exit(0);

default:

printf("Invalid choice\n");}}return 0;}

**RESULT:**



**Practical-6: Array implementation using circular queue**

1. **ALOGRITHM INTIALIZATION(QUEUE Q)**

BEGIN : CQ.REAR=Size-1 CQ.FRONT=Size-1

END;

⦁**ALGORITHM ENQUEUE(CQUEUE CQ,KEY)**

BEGIN:

IF (CQ.REAR+1)%SIZE==CQ.FRONT THEN

write(Queue overflows) Exit(1)

CQ.REAR=(CQ.REAR+1)%SIZE

CQ.item[CQ.REAR]=key

END;

⦁**ALGORITHM DEQUEUE(CQUEUE CQ)**

BEGIN:

IF CQ.REAR==CQ.FRONT

write( queue overflow) Exit(1)

CQ.FRONT=(CQ.FRONT+1)%Size x=CQ.item[CQ.FRONT] RETURN(x)

END;

For all the above ALGORITHM

**Time Complexity: ϴ (1) Space Complexity: ϴ (1)**

# **Implementation**

#include <stdio.h> #define SIZE 5

int items[SIZE];

int front = -1, rear = -1; int isFull() {

if ((front == rear + 1) || (front == 0 && rear == SIZE - 1))

return 1; return 0;}int isEmpty() {

if (front == -1) return 1;

return 0;}void enQueue(int element) { if (isFull())

printf("\n Queue is full!! \n"); else {if (front == -1) front = 0; rear = (rear + 1) % SIZE; items[rear] = element;

printf("\n Inserted -> %d", element);}}int deQueue() { int element;

if (isEmpty()) {

printf("\n Queue is empty !! \n"); return (-1);

} else {

element = items[front]; if (front == rear) {

front = -1;

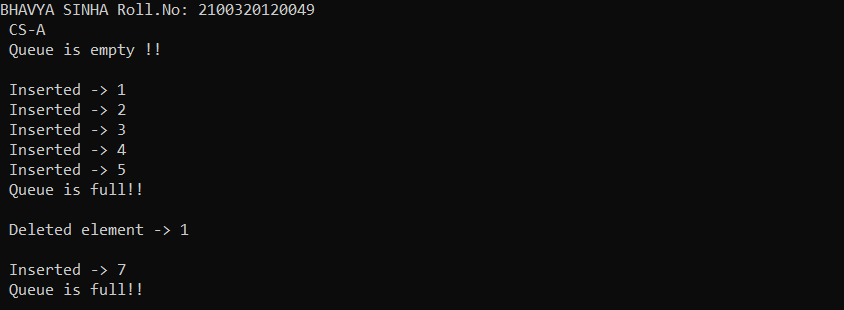
rear = -1;}else {front = (front + 1) % SIZE;}printf("\n Deleted element -> %d \n", element); return (element);}}

int main() {

printf("ARYAN CHOUDHARYRoll.No: 2100320120034\n CS-A"); deQueue();

enQueue(1); enQueue(2); enQueue(3); enQueue(4); enQueue(5); enQueue(6); display(); deQueue(); display(); enQueue(7); display(); enQueue(8); return 0;

}



PROBLEM STATEMENT 7: PROGRAM FOR LINKED LIST IMPLEMENTATION OF STACKS

ALGORITHIM

1. Define a struct for the linked list node, which contains two fields: **data** to store the value of the element and **next** to point to the next node in the list.
2. Define a struct for the stack, which contains a pointer to the top of the stack, initialized to **NULL**.
3. To push an element onto the stack: a. Allocate memory for a new node. b. Set the **data** field of the new node to the value of the element. c. Set the **next** field of the new node to point to the current top of the stack. d. Set the top of the stack to point to the new node.
4. To pop an element from the stack: a. Check if the stack is empty by checking if the top pointer is **NULL**. b. If the stack is not empty, store the value of the top node's **data** field in a variable. c. Set the top of the stack to point to the next node in the list. d. Free the memory allocated for the top node. e. Return the value of the stored variable.
5. To check if the stack is empty, simply check if the top pointer is **NULL**

IMPLEMENATATION

#include <stdio.h>

#include <stdlib.h>

typedef struct node {

int data;

struct node\* next;

} node\_t;

typedef struct stack {

node\_t\* top;

} stack\_t;

void push(stack\_t\* stack, int element) {

node\_t\* new\_node = (node\_t\*) malloc(sizeof(node\_t));

new\_node->data = element;

new\_node->next = stack->top;

stack->top = new\_node;}

int pop(stack\_t\* stack) {

if (stack->top == NULL) {

printf("Stack underflow\n");

return -1;}

int element = stack->top->data;

node\_t\* temp = stack->top;

stack->top = stack->top->next;

free(temp);

return element;}

int is\_empty(stack\_t\* stack) {

return (stack->top == NULL);}

int main() {

printf("Bhavya Sinha\n");

printf("2100320120032\n");

printf("CS\_A\n");

stack\_t\* stack = (stack\_t\*) malloc(sizeof(stack\_t));

stack->top = NULL;

push(stack, 10);

push(stack, 20);

push(stack, 30);

printf("%d\n", pop(stack));

printf("%d\n", pop(stack));

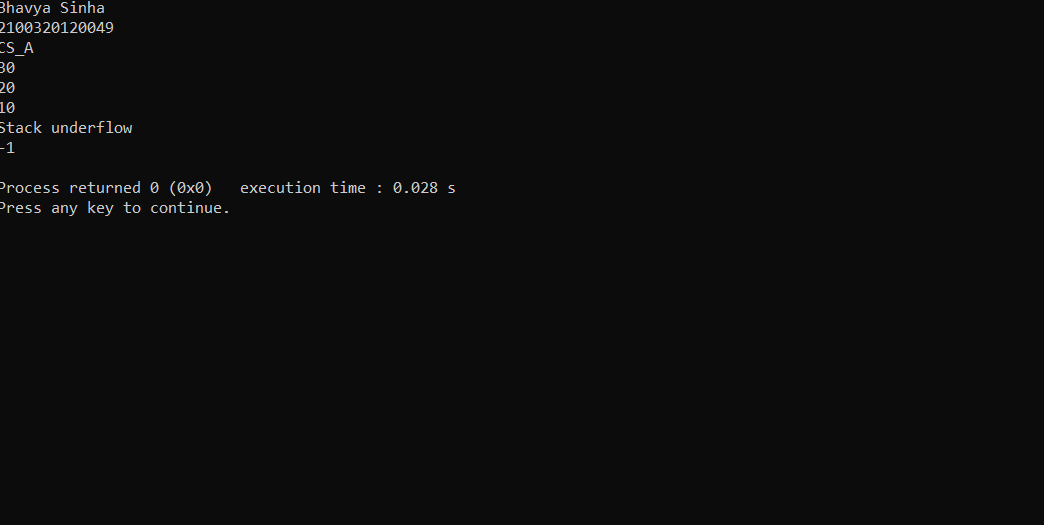
printf("%d\n", pop(stack));

printf("%d\n", pop(stack)); // Should print "Stack underflow"

free(stack);

return 0;}

RESULT



**PRACTICAL 8: Program for linked list implementation of queue.**

**1.ALGORITHM / FLOW CHART:**

**ALGORITHM Initialize(FRONT,REAR)**

**BEGIN:**

**REAR=NULL**

**FRONT=NULL**

**END;**

TIME COMPLEXITY: **ϴ** (1)

SPACE COMPLEXITY: **ϴ** (1)

**ALGORITHM Empty(FRONT)**

**BEGIN:**

**IF(FRONT==NULL) THEN**

**RETURN TRUE**

**ELSE**

**RETURN FALSE**

**END;**

TIME COMPLEXITY: **ϴ** (1)

SPACE COMPLEXITY: **ϴ** (1)

**ALGORITHM ENQUEUE(FRONT,REAR,x)**

**BEGIN:**

IF(REAR==NULL) THEN

insertbeg(REAR,x)

FRONT=REAR

ELSE

insertsafter(REAR,x)

REAR=REAR(next)

**END;**

TIME COMPLEXITY: **ϴ** (1)

SPACE COMPLEXITY: **ϴ** (1)

**ALGORITHM Dequeue(FRONT,REAR)**

**BEGIN:**

IF(FRONT==NULL) THEN

write("void deletion")

exit(1)

**ELSE**

x=delbeg(FRONT)

IF(FRONT==NULL)

REAR=NULL

RETURN x

**END;**

TIME COMPLEXITY: **ϴ** (1)

SPACE COMPLEXITY: **ϴ** (1)

**ALGORITHM Traverse(FRONT)**

**BEGIN:**

WHILE FRONT!=NULL

write(FRONT(info) )

FRONT=FRONT(next)

**END;**

TIME COMPLEXITY: **ϴ** (1)

SPACE COMPLEXITY: **ϴ** (1)

**2.IMPLEMENTATION:**

**#include<stdio.h>**

**#include<stdlib.h>**

**struct node**

**{int data;**

**struct node \*next;};**

**struct node \*front;**

**struct node \*rear;**

**void insert();**

**void delete();**

**void display();**

**void main ()**

**{ printf("Aryan Choudhary Roll.No: 2100320120034\n CS-A\n");**

**int choice;**

**while(choice != 4)**

**{ printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");**

**printf("\nEnter your choice ?");**

**scanf("%d",& choice);**

**switch(choice)**

**{ case 1:**

**insert();**

**break;**

**case 2:**

**delete();**

**break;**

**case 3:**

**display();**

**break;**

**case 4:**

**exit(0);**

**break;**

**default:**

**printf("\nEnter valid choice??\n");}} }**

**void insert()**

**{ struct node \*ptr;**

**int item;**

**ptr = (struct node \*) malloc (sizeof(struct node));**

**if(ptr == NULL)**

**{ printf("\nOVERFLOW\n");**

**return;}**

**else**

**{ printf("\nEnter value?\n");**

**scanf("%d",&item);**

**ptr -> data = item;**

**if(front == NULL)**

**{ front = ptr;**

**rear = ptr;**

**front -> next = NULL;**

**rear -> next = NULL; }**

**else**

**{ rear -> next = ptr;**

**rear = ptr;**

**rear->next = NULL;}}}**

**void delete ()**

**{ struct node \*ptr;**

**if(front == NULL)**

**{ printf("\nUNDERFLOW\n");**

**return;}**

**else**

**{ ptr = front;**

**front = front -> next;**

**free(ptr);}}**

**void display()**

**{ struct node \*ptr;**

**ptr = front;**

**if(front == NULL)**

**{**

**printf("\nEmpty queue\n"); }**

**else**

**{printf("\nprinting values .....\n");**

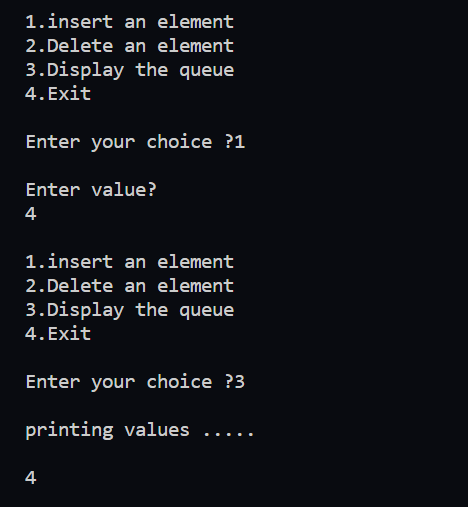
**while(ptr != NULL)**

**{**

**printf("\n%d\n",ptr -> data);**

**ptr = ptr -> next;}}}**

**3.** Result /Output:



**Practical 9:Write a program in c to implement circular queue using linked list.**

**Algorithm:**

**ALGORITHM ENQUEUE(x)**

**BEGIN:**

**q=getnode()**

IF(REAR==NULL) THEN

q(next)=q

ELSE

q(next)=REAR(next)

REAR(next)=q

REAR=q

**END;**

TIME COMPLEXITY: **ϴ** (1)

SPACE COMPLEXITY: **ϴ** (1)

**ALGORITHM Dequeue()**

**BEGIN:**

IF(REAR==NULL) THEN

write("void deletion")

exit(1)

x=REAR.next.data

IF REAR.next==REAR)

REAR=NULL

ELSE

REAR.next=REAR.next.next

RETURN x

**END;**

TIME COMPLEXITY: **ϴ** (1)

SPACE COMPLEXITY: **ϴ** (1)

**Implementation:**

#include<stdio.h>

#include<stdlib.h>

struct node{

int data;

struct node\* next;

};

struct node \*f = NULL;

struct node \*r = NULL;

void enqueue(int d) {

struct node\* n;

n = (struct node\*)malloc(sizeof(struct node));

n->data = d;

n->next = NULL;

if((r==NULL)&&(f==NULL)){

f = r = n;

r->next = f;

}

else{

r->next = n;

r = n;

n->next = f;

}}

void dequeue(){

struct node\* t;

t = f;

if((f==NULL)&&(r==NULL))

printf("\nQueue is Empty");

else if(f == r){

f = r = NULL;

free(t);

}

else{

f = f->next;

r->next = f;

free(t);

}}

void print(){

struct node\* t;

t = f;

if((f==NULL)&&(r==NULL))

printf("\nQueue is Empty");

else{

do{

printf("\n%d",t->data);

t = t->next;

}while(t != f);

}}

int main()

{ printf("Bhavya Sinha\n");printf("2100320120032 CS-A\n");

int opt,n,i,data;

printf("Enter Your Choice:-");

do{

printf("\n\n1.for Insert the Data in Queue\n2.for show the Data in Queue\n3.for Delete the data from the Queue\n0.for Exit");

scanf("%d",&opt);

switch(opt){

case 1:

printf("\nEnter the number of data");

scanf("%d",&n);

printf("\nEnter your data");

i=0;

while(i<n){

scanf("%d",&data);

enqueue(data);

i++;

}

break;

case 2:

print();

break;

case 3:

dequeue();

break;

case 0:

break;

default:

printf("\nIncorrect Choice");

}

}while(opt!=0);

return 0;}

Output:

Bhavya Sinha

2100320120032 CS-A

Enter Your Choice:-

1.for Insert the Data in Queue

2.for show the Data in Queue

3.for Delete the data from the Queue

0.for Exit 1

Enter the number of data5

Enter your data2

4

3

7

8

1.for Insert the Data in Queue

2.for show the Data in Queue

3.for Delete the data from the Queue

0.for Exit2

2

4

3

7

8

1.for Insert the Data in Queue

2.for show the Data in Queue

3.for Delete the data from the Queue

0.for Exit 3

1.for Insert the Data in Queue

2.for show the Data in Queue

3.for Delete the data from the Queue

0.for Exit

0

Process returned 0 (0x0) execution time : 48.490 s

Press any key to continue.

**Practical-10: Linked list implementation of priority queue**

# **Algorithm:**

1. Define a Node struct to represent each node in the priority queue. The struct should contain the node's value and priority, as well as a pointer to the next node in the list.

2. Define a function insert\_node that takes a pointer to a pointer to the head of the linked list, an integer value, and an integer priority. This function should create a new node with the given value and priority, and insert it into the list in the correct position based on its priority.

3. Define a function remove\_node that takes a pointer to a pointer to the head of the linked list, removes the node with the highest priority from the list, and returns its value.

4. Define a function print\_priority\_queue that takes a pointer to the head of the linked list and prints the contents of the priority queue in order of decreasing priority.

5. To insert a node into the priority queue, call the insert\_node function with the head pointer, value, and priority as arguments.

6. To remove a node from the priority queue, call the remove\_node function with the head pointer as an argument.

7. To print the contents of the priority queue, call the print\_priority\_queue function with the head pointer as an argument

**Implementation:**

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

struct Node

{int value; int priority;

struct Node \*next;};void insert\_node(struct Node \*\*head, int value, int priority)

{struct Node \*new\_node = (struct Node \*)malloc(sizeof(struct Node)); new\_node->value = value;

new\_node->priority = priority;

if (\*head == NULL || priority < (\*head)->priority)

{new\_node->next = \*head;

\*head = new\_node;}

else

{struct Node \*current\_node = \*head;

while (current\_node->next != NULL && current\_node->next->priority <= priority)

{current\_node = current\_node->next;}

new\_node->next = current\_node->next; current\_node->next = new\_node;}}

int remove\_node(struct Node \*\*head)

{if (\*head == NULL)

{printf("Error: queue is empty.\n"); return -1;}

struct Node \*node\_to\_remove = \*head; int value = node\_to\_remove->value;

\*head = node\_to\_remove->next; free(node\_to\_remove);

return value;

}void print\_priority\_queue(struct Node \*head)

{if (head == NULL)

{printf("Priority queue is empty.\n"); return;}printf("Priority queue: "); while (head != NULL)

{printf("(%d,%d) ", head->value, head->priority); head = head->next;}

printf("\n");}

int main()

{ printf("Name:Bhavya Sinha\tRoll NO:2100320120032\nCS-A\n");

struct Node \*head = NULL; insert\_node(&head, 1, 2);

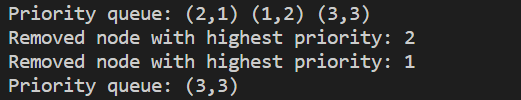
insert\_node(&head, 2, 1);

insert\_node(&head, 3, 3); print\_priority\_queue(head);

printf("Removed node with highest priority: %d\n", remove\_node(&head)); printf("Removed node with highest priority: %d\n", remove\_node(&head)); print\_priority\_queue(head);

return 0;}

# **RESULT:**



**Practical-11:** Implementation of Tree Traversal

**Algorithm:**

**ALGORITHM PreorderTraversal(root)**

**BEGIN:**

IF root != NULL THEN

WRITE(Data(root))

PreorderTraversal(Left(root))

PreorderTraversal(Right(root))

**END;**

TIME COMPLEXITY: **ϴ** (N)

SPACE COMPLEXITY: **ϴ** (LogN) for balanced trees

**ALGORITHM PostorderTraversal(root)**

**BEGIN:**

IF root != NULL THEN

PostorderTraversal(Left(root))

PostorderTraversal(Right(root))

WRITE(Data(root))

**END;**

TIME COMPLEXITY: **ϴ** (N)

SPACE COMPLEXITY: **ϴ** (LogN) for balanced trees

**ALGORITHM InorderTraversal(root)**

**BEGIN:**

WHILE root != NULL THEN

InorderTraversal(Left(root))

WRITE(Data(root))

InorderTraversal(Right(root))

**END;**

TIME COMPLEXITY: **ϴ** (N)

SPACE COMPLEXITY: **ϴ** (LogN) for balanced trees

**Implementation:**

**#include <stdio.h> #include <stdlib.h> struct node {**

**int data;**

**struct node\* left; struct node\* right;**

**};**

**struct node\* newNode(int data){ struct node\* node**

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

**node->left = NULL; node->right = NULL; return (node);**

**}**

**void printInorder(struct node\* node){ if (node == NULL)**

**return; printInorder(node->left);**

**printf("%d ", node->data); printInorder(node->right);**

**}**

**void printPreorder(struct node\* node){ if (node == NULL)**

**return;**

**printf("%d ", node->data); printPreorder(node->left); printPreorder(node->right);**

**}**

**void printPostorder(struct node\* node)**

**{**

**if (node == NULL) return;**

**printPostorder(node->left); printPostorder(node->right); printf("%d ", node->data);**

**}**

**int main(){**

**struct node\* root = newNode(1); root->left = newNode(2);**

**root->right = newNode(3); root->left->left = newNode(4);**

**root->left->right = newNode(5);**

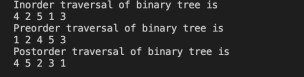
**printf("ARYAN ARORARoll.No: 2100320120034\n CS-A"); printf("\nInorder traversal of binary tree is \n"); printInorder(root);**

**printf("\nPreorder traversal of binary tree is \n"); printPreorder(root);**

**printf("\nPostorder traversal of binary tree is \n"); printPostorder(root);**

**getchar(); return 0;**

**}**



**Practical 12: Write a program in c to implement binary search tree.**

**Algorithm:**

**ALGORITHM BSTInsert(Tree,key)**

**BEGIN:**

P = Tree

Q = Null

R = MakeNode(key)

WHILE P!=NULL Do

IF key &lt; Data(P)

Q = P

P = Left(P)

ELSE

Q = P

P = Right(P)

IF key &lt; Data(Q) THEN

Left(Q) = R

Father(R) = Q

ELSE

Right(Q) = R

Father(R) = Q

**END;**

**Time Complexity: ϴ(LogN) if Tree is balanced**

**Space Complexity: ϴ(1)**

Implementation:

#include <stdio.h> #include <stdlib.h> struct node {

int key;

struct node \*left, \*right;

};struct node\* newNode(int item){ struct node\* temp

= (struct node\*)malloc(sizeof(struct node)); temp->key = item;

temp->left = temp->right = NULL; return temp;}void inorder(struct node\* root)

{if (root != NULL) { inorder(root->left); printf("%d \n", root->key); inorder(root->right);

}}

struct node\* insert(struct node\* node, int key){ if (node == NULL)

return newNode(key); if (key < node->key)

node->left = insert(node->left, key); else if (key > node->key)

node->right = insert(node->right, key); return node;}

int main(){

struct node\* root = NULL; root = insert(root, 50);

**printf("ARYAN ARORARoll.No: 2100320120034\n CS-A");**

insert(root, 30);

insert(root, 20);

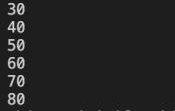
insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80); inorder(root); return 0;

}



**Practical-13: insertion and Deletion in BST**

## **Algorithm:**

|  |  |
| --- | --- |
| 1. If the root is NULL, return NULL. 2. If the key to be deleted is less than the data of the root node, recursively delete the key from the left subtree and update the left child of the root. 3. If the key to be deleted is greater than the data of the root node, recursively delete the key from the right subtree and update the right child of the root. 4. If the key to be deleted is equal to the data of the root node, there are three cases: | |
|  | 1. If the root has no children or only one child, then free the root and return NULL or the child node as appropriate. 2. If the root has two children, find the inorder successor (the node with the smallest key in the right subtree) of the root, copy its data to the root, and recursively delete the inorder successor from the right subtree. 3. If the root has two children, but the right subtree is empty, find the inorder predecessor (the node with the largest key in the left subtree) of the root, copy its   data to the root, and recursively delete the inorder predecessor from the left subtree. |
| 5. Return the updated root node. | |

**Implementation:**

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

// Define the structure of a binary tree node struct node

{

int data;

struct node \*left;

struct node \*right;

};

// Function to create a new binary tree node with the given data struct node \*newNode(int data)

{

struct node \*newNode = (struct node \*)malloc(sizeof(struct node));

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Function to find the minimum value in a BST struct node \*findMin(struct node \*root)

{

while (root->left != NULL)

{

root = root->left;

}

return root;

}

// Function to delete a node with the given key from a BST struct node \*deleteNode(struct node \*root, int key)

{

if (root == NULL)

{

return root;

}

if (key < root->data)

{

root->left = deleteNode(root->left, key);

}

else if (key > root->data)

{

root->right = deleteNode(root->right, key);

}

else

{

// Case 1: No child or one child

if (root->left == NULL)

{

struct node \*temp = root->right;

free(root);

return temp;

}

else if (root->right == NULL)

{

struct node \*temp = root->left;

free(root);

return temp;

}

struct node \*temp = findMin(root->right);

root->data = temp->data;

root->right = deleteNode(root->right, temp->data);

}

return root;

}

// Function to print the inorder traversal of a BST void inorderTraversal(struct node \*root)

{

if (root != NULL)

{

inorderTraversal(root->left);

printf("%d ", root->data);

inorderTraversal(root->right);

}

}

// Driver code int main()

{

printf("Name:Bhavya Sinha\tRoll NO:2100320120032\nCS-A\n");

struct node \*root = NULL;

root = newNode(5);

root->left = newNode(3);

root->right = newNode(7);

root->left->left = newNode(2);

root->left->right = newNode(4);

root->right->left = newNode(6);

root->right->right = newNode(8);

printf("Inorder traversal of the original BST: ");

inorderTraversal(root);

printf("\n");

root = deleteNode(root, 3);

printf("Inorder traversal of the BST after deletion: ");

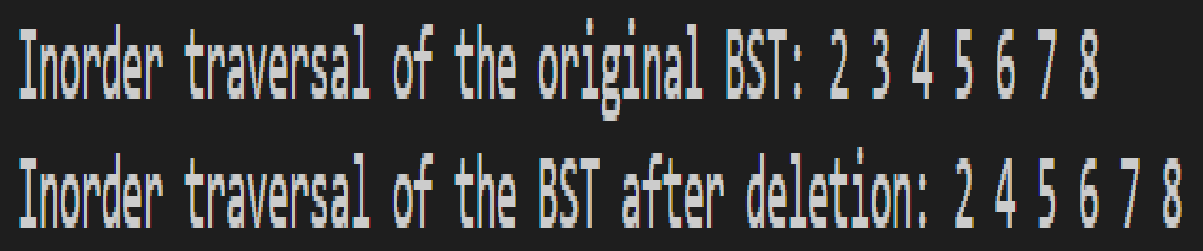
inorderTraversal(root);

printf("\n");

return 0;

}

## **RESULT:**



**Practical-17: Program for BFS on a Graph**

**Algorithm:**

create a queue Q

mark v as visited and put v into Q

while Q is non-empty

remove the head u of Q

mark and enqueue all (unvisited) neighbours of u

**Implementation:**

#include <stdio.h>

#include <stdlib.h>

#define SIZE 40

struct queue {

int items[SIZE];

int front;

int rear;};

struct queue\* createQueue();

void enqueue(struct queue\* q, int);

int dequeue(struct queue\* q);

void display(struct queue\* q);

int isEmpty(struct queue\* q);

void printQueue(struct queue\* q);

struct node {int vertex;

struct node\* next;};

struct node\* createNode(int);

struct Graph {

int numVertices;

struct node\*\* adjLists;

int\* visited;

};

void bfs(struct Graph\* graph, int startVertex) {

struct queue\* q = createQueue();

graph->visited[startVertex] = 1;

enqueue(q, startVertex);

while (!isEmpty(q)) {

printQueue(q);

int currentVertex = dequeue(q);

printf("Visited %d\n", currentVertex);

struct node\* temp = graph->adjLists[currentVertex]; while (temp) {

int adjVertex = temp->vertex;

if (graph->visited[adjVertex] == 0) {

graph->visited[adjVertex] = 1;

enqueue(q, adjVertex);}

temp = temp->next;}}}

struct node\* createNode(int v) {

struct node\* newNode = malloc(sizeof(struct node));

newNode->vertex = v;

newNode->next = NULL;

return newNode;

}struct Graph\* createGraph(int vertices) {

struct Graph\* graph = malloc(sizeof(struct Graph));

graph->numVertices = vertices;

graph->adjLists = malloc(vertices \* sizeof(struct node\*)); graph->visited = malloc(vertices \* sizeof(int));

int i;

for (i = 0; i < vertices; i++) {

graph->adjLists[i] = NULL;

graph->visited[i] = 0;}return graph;}

void addEdge(struct Graph\* graph, int src, int dest) { // Add edge from src to dest

struct node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

newNode = createNode(src);

newNode->next = graph->adjLists[dest];

graph->adjLists[dest] = newNode;}

struct queue\* createQueue() {

struct queue\* q = malloc(sizeof(struct queue));

q->front = -1;

q->rear = -1;

return q;}

int isEmpty(struct queue\* q) {

if (q->rear == -1)

return 1;

else

return 0;}

void enqueue(struct queue\* q, int value) {

if (q->rear == SIZE - 1)

printf("\nQueue is Full!!");

else {if (q->front == -1)

q->front = 0;

q->rear++;

q->items[q->rear] = value;}}

int dequeue(struct queue\* q) {

int item;

if (isEmpty(q)) {

printf("Queue is empty");

item = -1;} else {

item = q->items[q->front];

q->front++;

if (q->front > q->rear) {

printf("Resetting queue ");

q->front = q->rear = -1;}}return item;}

void printQueue(struct queue\* q) {

int i = q->front;

if (isEmpty(q)) {

printf("Queue is empty");} else {

printf("\nQueue contains \n");

for (i = q->front; i < q->rear + 1; i++) {

printf("%d ", q->items[i]);}}}

int main() {

printf("Aryan AroraRoll.No: 2100320120034\n CS-A\n");

struct Graph\* graph = createGraph(6);

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 2);

addEdge(graph, 1, 4);

addEdge(graph, 1, 3);

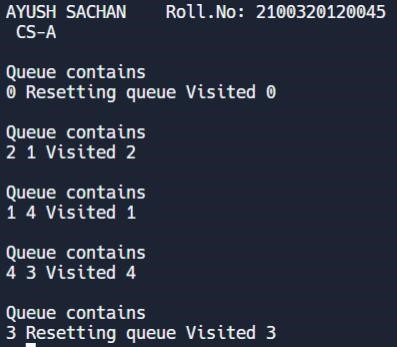
addEdge(graph, 2, 4);

addEdge(graph, 3, 4);

bfs(graph, 0);

return 0;}

**RESULT**:



**Practical-18 : DFS search on a graph**

## **Algorithm:**

|  |  |
| --- | --- |
| 1. Mark the starting node as visited and push it onto a stack. 2. While the stack is not empty, do the following: | |
|  | * Pop the top node from the stack. * For each unvisited neighbor of the node, mark it as visited and push it onto the stack. |
| 3. When the stack is empty, the search is complete. | |

**Implementation:**

#include <stdio.h>

#include <stdlib.h>

int vis[100];

struct Graph

{int V;

int E;

int \*\*Adj;};

struct Graph \*adjMatrix()

{struct Graph \*G = (struct Graph \*)

malloc(sizeof(struct Graph));

if (!G)

{ printf("Memory Error\n");

return NULL;}

G->V = 7;

G->E = 7;

G->Adj = (int \*\*)malloc((G->V) \* sizeof(int \*));

for (int k = 0; k < G->V; k++)

{G->Adj[k] = (int \*)malloc((G->V) \* sizeof(int));

}

for (int u = 0; u < G->V; u++)

{ for (int v = 0; v < G->V; v++)

{ G->Adj[u][v] = 0;}}

G->Adj[0][1] = G->Adj[1][0] = 1;

G->Adj[0][2] = G->Adj[2][0] = 1;

G->Adj[1][3] = G->Adj[3][1] = 1;

G->Adj[1][4] = G->Adj[4][1] = 1;

G->Adj[1][5] = G->Adj[5][1] = 1;

G->Adj[1][6] = G->Adj[6][1] = 1;

G->Adj[6][2] = G->Adj[2][6] = 1;

return G;}

void DFS(struct Graph \*G, int u)

{ vis[u] = 1;

printf("%d ", u);

for (int v = 0; v < G->V; v++)

{ if (!vis[v] && G->Adj[u][v])

{ DFS(G, v);}}}

void DFStraversal(struct Graph \*G)

{for (int i = 0; i < 100; i++)

{ vis[i] = 0;}

for (int i = 0; i < G->V; i++)

{if (!vis[i])

{ DFS(G, i);}}}

void main()

{printf("Name:Bhavya Sinha\tRoll NO:2100320120032\nCS-A\n");

printf("DFS search is:\n");

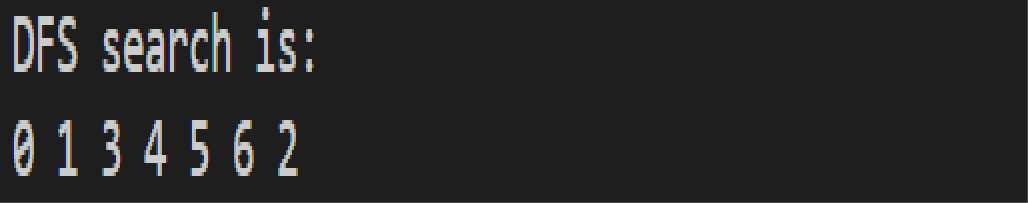
struct Graph \*G;

G = adjMatrix();

DFStraversal(G);

}

## **RESULT:**



**PRACTICAL 22: Program for warshall's algorithm for APSP.**

**1.ALGORITHM / FLOW CHART:**

**ALGORITHM APSPFloydWarshall ( W [ ] [ ], N)**

**BEGIN:**

FOR i ← 1 To N Do

FOR i ← 1 To N DO

IF W [i] [ j] = = 0

IF i != j THEN

W [i] [j]= ∞

FOR k ← 1 To N Do

FOR i ← 1 To N DO

FOR j ← 1 To N DO

W [i] [j] ← Min (W [i] [j] , W [i] [k]+ W [k] [j])

**END ;Time Complexity: ϴ(N3)**

**Space Complexity: ϴ(1)**

**2.IMPLEMENTATION:**

**#include <stdio.h>**

**#define V 4**

**#define INF 99999**

**void printSolution(int dist[][V]);**

**void floydWarshall(int dist[][V])**

**{int i, j, k;**

**for (k = 0; k < V; k++) {**

**for (i = 0; i < V; i++) {**

**for (j = 0; j < V; j++) {**

**if (dist[i][k] + dist[k][j] < dist[i][j])**

**dist[i][j] = dist[i][k] + dist[k][j];}}}**

**printSolution(dist);}**

**void printSolution(int dist[][V])**

**{printf("The following matrix shows the shortest distances"**

**" between every pair of vertices \n");**

**for (int i = 0; i < V; i++) {**

**for (int j = 0; j < V; j++) {**

**if (dist[i][j] == INF)**

**printf("%7s", "INF");**

**else**

**printf("%7d", dist[i][j]);}**

**printf("\n");}}**

**int main()**

**{printf("Aryan Choudhary Roll.No: 2100320120034\n CS-A\n");**

**int graph[V][V] = { { 0, 5, INF, 10 },**

**{ INF, 0, 3, INF },**

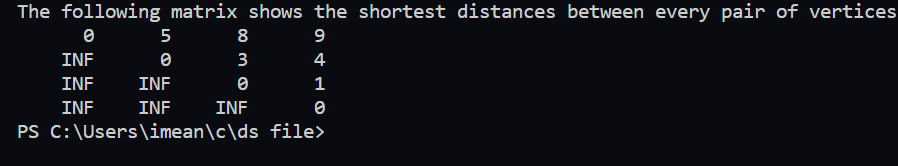
**{ INF, INF, 0, 1 },**

**{ INF, INF, INF, 0 } };**

**floydWarshall(graph);**

**return 0;}**

**3.** Result /Output:



**LAB 21**

**Implementation of single source shortest path algorithm**

**ALGORITHM SSSPDijkstra (G , W [ ] [ ], S)**

**BEGIN:**

Priority Queue PQ

Initialize (PQ)

For all U ϵ V [G] DO

D [U] ← ∞

Π[U] ← NIL

PQ Insert (PQ , U)

D [S] ← 0

WHILE ! Empty (PQ) DO

U ← ExtractMin(PQ)

For All V ϵ Adj [U] AND V ϵ PQ

IF d[V] <d[U] +W [U][V]

d[V] ← d[U] +W [U][V]

Π[V] ← U

**END ;**

**Time Complexity: O(|E|+|V|Log |V|)**

**Space Complexity: ϴ(|V|)**

**Implementation :**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

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

int minDistance(int dist[], int sptSet[]) {

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++)

if (sptSet[v] == 0 && 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];

int sptSet[V];

for (int i = 0; i < V; i++) {

dist[i] = INT\_MAX;

sptSet[i] = 0;} dist[src] = 0;

for (int count = 0; count < V-1; count++) {

int u = minDistance(dist, sptSet);

sptSet[u] = 1;

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, 2, 0, 6, 0},

{2, 0, 3, 8, 5},

{0, 3, 0, 0, 7},

{6, 8, 0, 0, 9},

{0, 5, 7, 9, 0}};

printf("The given graph is : \n");

for (int i = 0; i < V; i++)

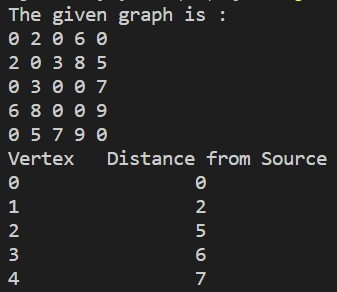
{

for (int j = 0; j < V; j++)

{printf("%d ", graph[i][j]);}printf("\n");} dijkstra(graph, 0);

return 0;}

**Output :**



**LAB 19**

**Implementation of Minimum cost spanning tree using Kruskal**

**ALGORITHM MST- Kruskal (G , W[ ][ ])**

**BEGIN:**

Sort all Edges In E According To their Weights

Set of Edges of MST E’← ∅

For all U ɛ V [G] DO

MakeSet ( U)

For Each Edge(U , V) ɛ E [G] DO

IF FindSet (U)! = FindSet (V)

Union(U, V)

Select(U, V) In E’

RETURN E’

**END ;Time Complexity: O(|E|+|V|Log |V|)**

**Space Complexity: ϴ(|V|)**

**Implementation :**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_EDGES 10

#define MAX\_VERTICES 6

int graph[MAX\_EDGES][3] = {

{3, 4, 9},

{3, 5, 4},

{4, 5, 1},

{999, 999, 999} // sentinel value to indicate end of edges

};

int parent[MAX\_VERTICES];

int rank[MAX\_VERTICES];

int find(int vertex) {

if (parent[vertex] != vertex) {parent[vertex] = find(parent[vertex]);} return parent[vertex];}

void union\_by\_rank(int root1, int root2) {

if (rank[root1] < rank[root2]) {

parent[root1] = root2;

} else if (rank[root1] > rank[root2]) {

parent[root2] = root1;

} else { parent[root2] = root1;

rank[root1]++;}}

void kruskal() {

int i, j, root1, root2, edge\_count = 0, min\_cost = 0;

for (i = 0; i < MAX\_VERTICES; i++) {

parent[i] = i;

rank[i] = 0;} // Sort edges by increasing order of weight

for (i = 0; i < MAX\_EDGES - 1; i++) {

for (j = i + 1; j < MAX\_EDGES; j++) {

if (graph[i][2] > graph[j][2]) {

int temp[3];

temp[0] = graph[i][0];

temp[1] = graph[i][1];

temp[2] = graph[i][2];

graph[i][0] = graph[j][0];

graph[i][1] = graph[j][1];

graph[i][2] = graph[j][2];

graph[j][0] = temp[0];

graph[j][1] = temp[1];

graph[j][2] = temp[2];}}} // Construct the minimum cost spanning tree

i = 0;

while (edge\_count < MAX\_VERTICES - 1) {

root1 = find(graph[i][0]);

root2 = find(graph[i][1])

if (root1 != root2) {

union\_by\_rank(root1, root2);

edge\_count++;

min\_cost += graph[i][2];} i++;}printf("Minimum cost = %d\n", min\_cost);}int main() {

printf("The given graph is : \n");

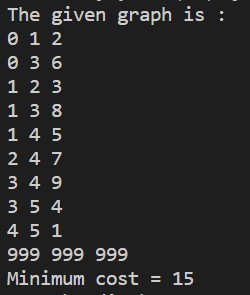
for (int i = 0; i < MAX\_EDGES; i++)

{for (int j = 0; j < 3; j++) {

printf("%d ", graph[i][j]);

}printf("\n");}kruskal();return 0;}

**Output :**



**LAB 20**

**Implementation of Minimum cost spanning tree using Prims**

**ALGORITHM MST Prims (G, W[ ][ ], R)**

**BEGIN:**

Priority Queue PQ

For all U ϵ V[G] Do

Key[U] ← ∞

Π[U] ← NIL

PQ Insert (PQ , U)

Key [R] ← 0

WHILE !Empty(PQ) Do

U ← ExtarctMin (PQ)

For all V ϵ Adj [U] AND V ϵ PQ DO

IF W[U][V] < Key [V] DO

Key [V] ← W [V][U]

Π[U] ← U

**END;**

**Time Complexity: O(|E|+|V|Log |V|)**

**Space Complexity: ϴ(|V|)**

**Implementation:**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

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

int minKey(int key[], int mstSet[]) {

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++)

if (mstSet[v] == 0 && 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];

int key[V];

int mstSet[V];

for (int i = 0; i < V; i++) {

key[i] = INT\_MAX;

mstSet[i] = 0;

}

key[0] = 0;

parent[0] = -1;

for (int count = 0; count < V-1; count++) {

int u = minKey(key, mstSet);

mstSet[u] = 1;

for (int v = 0; v < V; v++)

if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v])

parent[v] = u, key[v] = graph[u][v];

}printMST(parent, graph);}

int main() {

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}};

printf("The given graph is : \n");

for (int i = 0; i < V; i++)

{

for (int j = 0; j < V; j++)

{

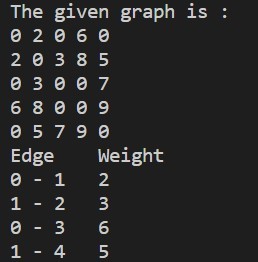
printf("%d ", graph[i][j]);}

printf("\n"); }

primMST(graph);

return 0;}

**Output :**



**LAB 14**

**Implementation of Max Heap**

**ALGORITHM MaxHeapIFy(A[],N)**

**BEGIN:**

FOR i=N/2 TO STEP-1 DO

Adjust(A,i,N)

**END;**

**ALGORITHM Adjust(A[],i,N)**

**BEGIN:**

WHILE 2\*i<=N DO

j=2\*i

IF j+1<=N THEN

IF A[j+1]>A[j]

j=j+1

IF A[j]>A[i] THEN

Exchange(A[j],A[i])

ELSE

BREAK

i=j

**END;**

#include <stdio.h>

#include <stdlib.h>

void heapify\_up(int heap[], int i) {

int parent = (i - 1) / 2;

if (heap[i] > heap[parent] && i > 0) {

int temp = heap[i];

heap[i] = heap[parent];

heap[parent] = temp;

heapify\_up(heap, parent);

}

}

void heapify\_down(int heap[], int heap\_size, int i) {

int largest = i;

int left\_child = 2 \* i + 1;

int right\_child = 2 \* i + 2;

if (left\_child < heap\_size && heap[left\_child] > heap[largest]) {

largest = left\_child;

}

if (right\_child < heap\_size && heap[right\_child] > heap[largest]) {

largest = right\_child;

}

if (largest != i) {

int temp = heap[i];

heap[i] = heap[largest];

heap[largest] = temp;

heapify\_down(heap, heap\_size, largest);

}

}

void insert(int heap[], int \*heap\_size, int value) {

heap[\*heap\_size] = value;

\*heap\_size += 1;

heapify\_up(heap, \*heap\_size - 1);

}

int extract\_max(int heap[], int \*heap\_size) {

int max = heap[0];

heap[0] = heap[\*heap\_size - 1];

\*heap\_size -= 1;

heapify\_down(heap, \*heap\_size, 0);

return max;

}

void build\_heap(int heap[], int n) {

for (int i = n / 2 - 1; i >= 0; i--) {

heapify\_down(heap, n, i);

}

}

void heap\_sort(int arr[], int n) {

int heap[n];

int heap\_size = 0;

for (int i = 0; i < n; i++) {

insert(heap, &heap\_size, arr[i]);

}

for (int i = n - 1; i >= 0; i--) {

arr[i] = extract\_max(heap, &heap\_size);

}

}

int main() {

printf(“Aryan AroraCSA 2100320120034\n”);

int heap[] = {7, 6, 5, 4, 3, 2, 1};

int n = sizeof(heap) / sizeof(heap[0]);

build\_heap(heap, n);

printf("Max heap: ");

for (int i = 0; i < n; i++) {

printf("%d ", heap[i]);

}

printf("\n");

heap\_sort(heap, n);

printf("Heap sorted array: ");

for (int i = 0; i < n; i++) {

printf("%d ", heap[i]);

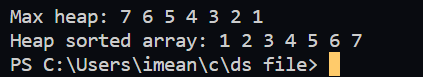
}

printf("\n");

return 0;

}

Output:



**LAB 15**

**Graph Memory Representation using Adjacency Matrix**

**ALGORITHM:**

1.Create a 2D array of size n x n, where n is the number of vertices in the graph.

2.Initialize all elements of the array to 0.

3.For each edge (u, v) in the graph, set the corresponding element in the array to 1, where u and v are the vertices connected by the edge.

4.If the graph is directed, set the element at row u and column v to 1, but not the element at row v and column u.

5.If the graph is weighted, store the weight of the edge in the corresponding element of the array, instead of 1.

#include <stdio.h>

#include <stdlib.h>

#define MAX\_VERTICES 100

int adj\_matrix[MAX\_VERTICES][MAX\_VERTICES];

int num\_vertices;

void init\_adj\_matrix() {

for (int i = 0; i < MAX\_VERTICES; i++) {

for (int j = 0; j < MAX\_VERTICES; j++) {

adj\_matrix[i][j] = 0;

}

}

}

void add\_edge(int u, int v, int weight) {

adj\_matrix[u][v] = weight;

}

void print\_adj\_matrix() {

printf("Adjacency matrix:\n");

printf(" ");

for (int i = 0; i < num\_vertices; i++) {

printf("%d ", i + 1);

}

printf("\n");

for (int i = 0; i < num\_vertices; i++) {

printf("%d |", i + 1);

for (int j = 0; j < num\_vertices; j++) {

printf("%d ", adj\_matrix[i][j]);

}

printf("\n");

}

}

int main() {

printf(“Aryan AroraCSA 2100320120034 \n”);

num\_vertices = 5;

init\_adj\_matrix();

add\_edge(0, 1, 1);

add\_edge(0, 3, 3);

add\_edge(1, 2, 2);

add\_edge(2, 0, 1);

add\_edge(2, 3, 1);

add\_edge(3, 4, 1);

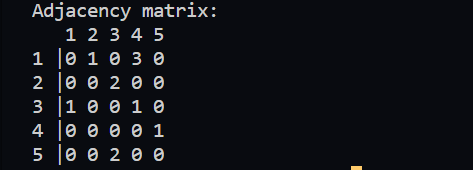
add\_edge(4, 2, 2);

print\_adj\_matrix();

return 0;

}

Output:



**LAB 16**

**Graph Memory Representation using Adjacency List**

**ALGORITHM:**

1.Create an array of size n, where n is the number of vertices in the graph. Each element in the array represents a vertex in the graph.

2.For each vertex, create a linked list to store its adjacent vertices.

3.For each edge (u, v) in the graph, add v to the linked list for u, and add u to the linked list for v (if the graph is undirected).

4.If the graph is directed, add v to the linked list for u only.

5.If the graph is weighted, store the weight of the edge along with the adjacent vertex in the linked list.

#include <stdio.h>

#include <stdlib.h>

struct node {

int vertex;

int weight;

struct node\* next;

};struct node\* create\_node(int v, int w) {

struct node\* new\_node = (struct node\*) malloc(sizeof(struct node));

new\_node->vertex = v;

new\_node->weight = w;

new\_node->next = NULL;

return new\_node;

}struct Graph {

int num\_vertices;

struct node\*\* adj\_lists;

};

struct Graph\* create\_graph(int n) {

struct Graph\* graph = (struct Graph\*) malloc(sizeof(struct Graph));

graph->num\_vertices = n;

graph->adj\_lists = (struct node\*\*) malloc(n \* sizeof(struct node\*));

for (int i = 0; i < n; i++) {

graph->adj\_lists[i] = NULL;

}

return graph;

}void add\_edge(struct Graph\* graph, int u, int v, int w) {

struct node\* new\_node = create\_node(v, w);

new\_node->next = graph->adj\_lists[u];

graph->adj\_lists[u] = new\_node;

if (graph->num\_vertices > v) {

new\_node = create\_node(u, w);

new\_node->next = graph->adj\_lists[v];

graph->adj\_lists[v] = new\_node;}}

Void print\_graph(struct Graph\* graph) {

printf("Adjacency List:\n");

for (int i = 0; i < graph->num\_vertices; i++) {

struct node\* temp = graph->adj\_lists[i];

printf("Vertex %d: ", i + 1);

while (temp) {

printf("%d (%d) -> ", temp->vertex + 1, temp->weight);

temp = temp->next;

}

printf("NULL\n");

}

}

int main() {

printf(“Aryan AroraCSA 2100320120034\n”);

int num\_vertices = 5;

struct Graph\* graph = create\_graph(num\_vertices);

add\_edge(graph, 0, 1, 1);

add\_edge(graph, 0, 3, 3);

add\_edge(graph, 1, 2, 2);

add\_edge(graph, 2, 0, 1);

add\_edge(graph, 2, 3, 1);

add\_edge(graph, 3, 4, 1);

add\_edge(graph, 4, 2, 2);

print\_graph(graph);

return 0;

}

Output:

