**Data Structure Lab**

**Q1. To implement stack using array.**

**#include<stdio.h>**

**int stack[100],choice,n,top,x,i;**

**void push(void);**

**void pop(void);**

**void display(void);**

**int main()**

**{**

**top=-1;**

**printf("\n Enter the size of STACK[MAX=100]:");**

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

**printf("\n\t STACK OPERATIONS USING ARRAY");**

**printf("\n\t--------------------------------");**

**printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");**

**do**

**{**

**printf("\n Enter the Choice:");**

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

**switch(choice)**

**{**

**case 1:**

**{**

**push();**

**break;**

**}**

**case 2:**

**{**

**break;**

**}**

**case 3:**

**{**

**display();**

**break;**

**}**

**case 4:**

**{**

**printf("\n\t EXIT POINT ");**

**break;**

**}**

**default:**

**{**

**printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");**

**}**

**}**

**}**

**while(choice!=4);**

**return 0;**

**}**

**void push()**

**{**

**if(top>=n-1)**

**{**

**printf("\n\tSTACK is over flow");**

**}**

**else**

**{**

**printf(" Enter a value to be pushed:");**

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

**top++;**

**stack[top]=x;**

**}**

**}**

**void pop()**

**{**

**if(top<=-1)**

**{**

**printf("\n\t Stack is under flow");**

**}**

**else**

**{**

**printf("\n\t The popped elements is %d",stack[top]);**

**top--;**

**}**

**}**

**void display()**

**{**

**if(top>=0)**

**{**

**printf("\n The elements in STACK \n");**

**for(i=top; i>=0; i--)**

**printf("\n%d",stack[i]);**

**printf("\n Press Next Choice");**

**}**

**else**

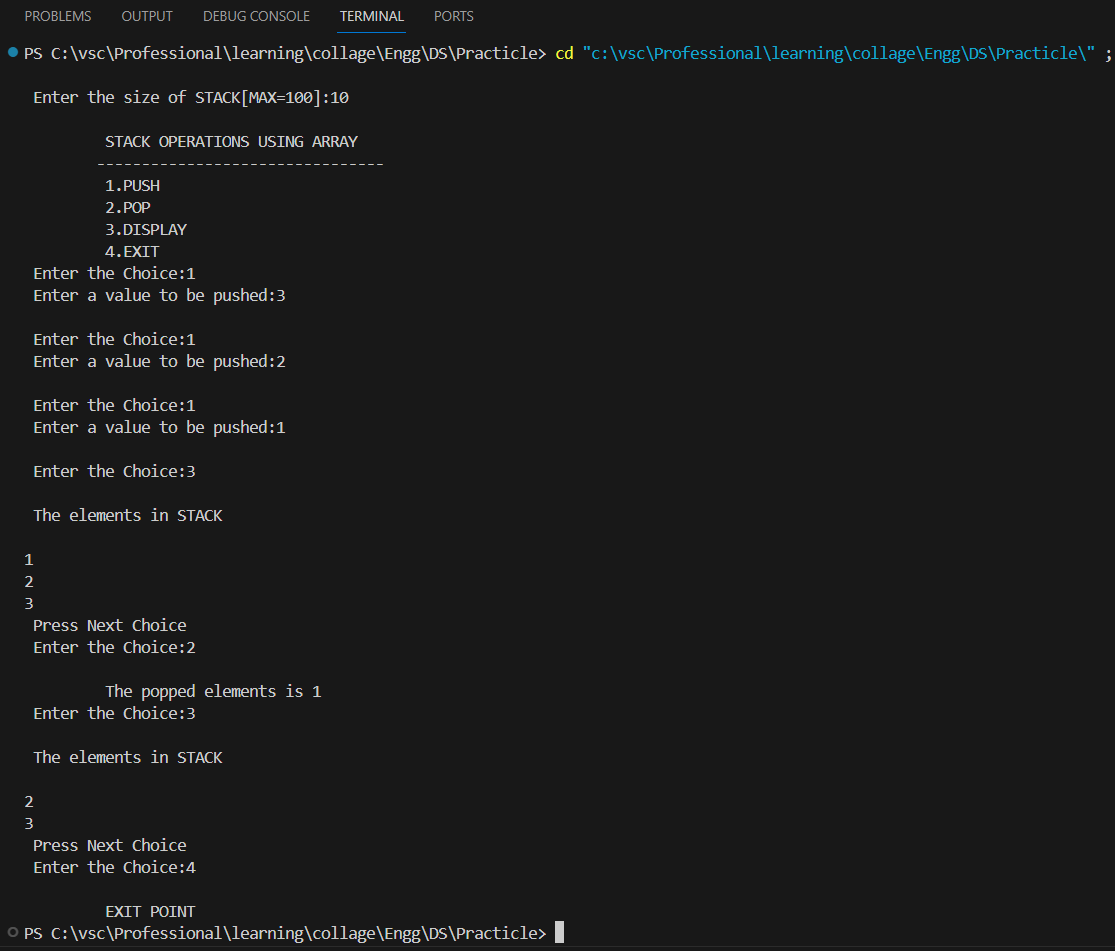
**{**

**printf("\n The STACK is empty");**

**}**

**}**

**Output:**

****

**2. To implement queue using array.**

**#include <stdio.h>**

**int queue[100], choice, n, front = -1, rear = -1, x;**

**void enqueue(void);**

**void dequeue(void);**

**void display(void);**

**int main() {**

**front = rear = -1;**

**printf("\nEnter the size of QUEUE (MAX=100):");**

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

**printf("\n\tQUEUE OPERATIONS USING ARRAY\n");**

**printf("\t--------------------------------\n");**

**printf("\t1. ENQUEUE\n");**

**printf("\t2. DEQUEUE\n");**

**printf("\t3. DISPLAY\n");**

**printf("\t4. EXIT\n");**

**do {**

**printf("\nEnter the Choice:");**

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

**switch (choice) {**

**case 1:**

**enqueue();**

**break;**

**case 2:**

**dequeue();**

**break;**

**case 3:**

**display();**

**break;**

**case 4:**

**printf("\n\tEXIT POINT\n");**

**break;**

**default:**

**printf("\n\tPlease Enter a Valid Choice (1/2/3/4)\n");**

**}**

**} while (choice != 4);**

**return 0;**

**}**

**void enqueue() {**

**if (rear == n - 1) {**

**printf("\n\tQUEUE is overflow\n");**

**} else {**

**if (front == -1) {**

**front = 0;**

**}**

**printf("Enter a value to be enqueued:");**

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

**rear++;**

**queue[rear] = x;**

**}**

**}**

**void dequeue() {**

**if (front == -1) {**

**printf("\n\tQUEUE is underflow\n");**

**} else {**

**printf("\n\tThe dequeued element is %d\n", queue[front]);**

**front++;**

**if (front > rear) {**

**front = rear = -1;**

**}**

**}**

**}**

**void display() {**

**if (front == -1) {**

**printf("\nQUEUE is Empty\n");**

**} else {**

**printf("\nQUEUE elements:\n");**

**for (int i = front; i <= rear; i++) {**

**printf("%d ", queue[i]);**

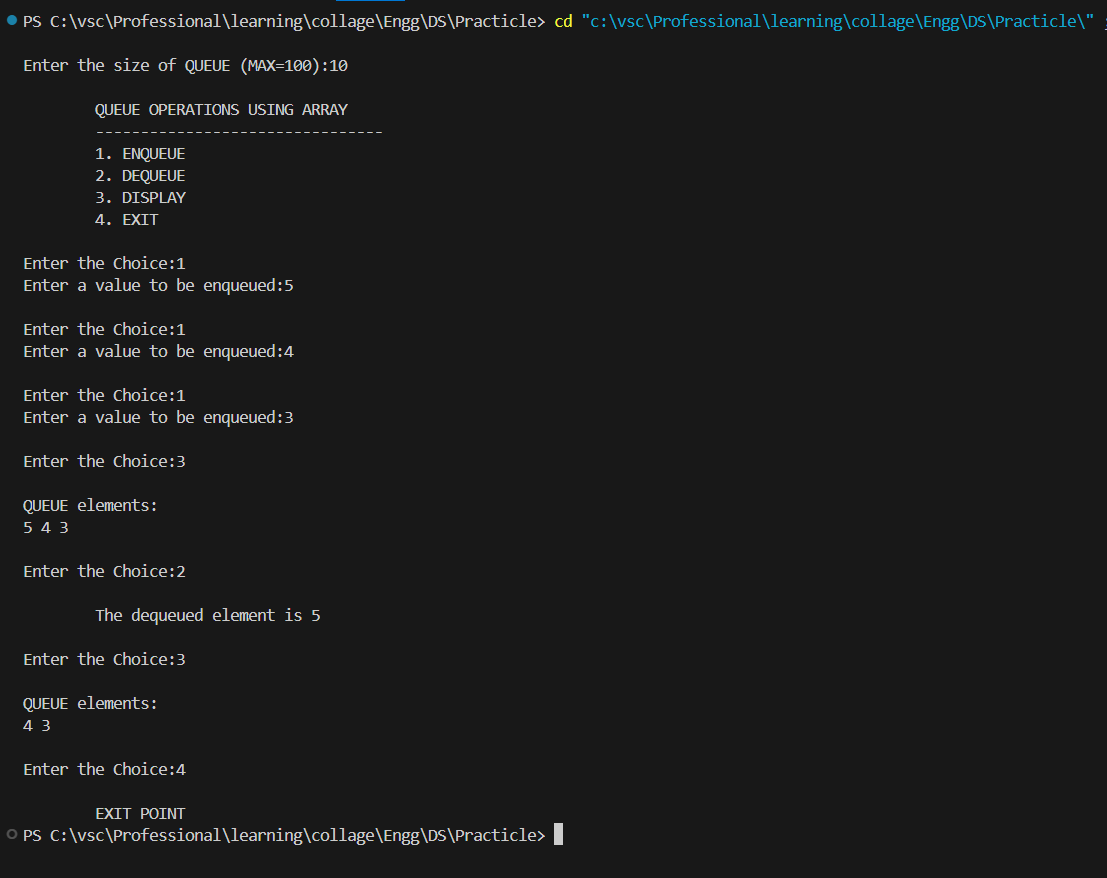
**}**

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

**}**

**}**

Output:



**3. To implement circular queue using array.**

**#include <stdio.h>**

**#include <stdlib.h>**

**#define MAX\_SIZE 100**

**int queue[MAX\_SIZE];**

**int front = -1, rear = -1, n; // n is the size of the queue**

**void enqueue(int x) {**

**if ((rear + 1) % MAX\_SIZE == front) {**

**printf("Queue overflow\n");**

**return;**

**}**

**// Handle the first element being enqueued**

**if (front == -1) {**

**front = 0;**

**}**

**rear = (rear + 1) % MAX\_SIZE; // Circular increment**

**queue[rear] = x;**

**printf("Inserted element: %d\n", x);**

**}**

**void dequeue() {**

**if (front == -1) {**

**printf("Queue underflow\n");**

**return;**

**}**

**printf("Dequeued element: %d\n", queue[front]);**

**if (front == rear) {  // Queue becomes empty after dequeue**

**front = rear = -1;**

**} else {**

**front = (front + 1) % MAX\_SIZE; // Circular increment**

**}**

**}**

**void display() {**

**if (front == -1) {**

**printf("Queue is empty\n");**

**return;**

**}**

**printf("Queue elements: ");**

**if (rear >= front) {**

**// Print elements from front to rear**

**for (int i = front; i <= rear; i++) {**

**printf("%d ", queue[i]);**

**}**

**} else {**

**// Wrap around for printing (front > rear)**

**for (int i = front; i < MAX\_SIZE; i++) {**

**printf("%d ", queue[i]);**

**}**

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

**printf("%d ", queue[i]);**

**}**

**}**

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

**}**

**int main() {**

**printf("Enter the size of the circular queue (MAX=%d): ", MAX\_SIZE);**

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

**int choice, value;**

**while (1) {**

**printf("\n1. Enqueue\n2. Dequeue\n3. Display\n4. Exit\n");**

**printf("Enter your choice: ");**

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

**switch (choice) {**

**case 1:**

**printf("Enter the value to enqueue: ");**

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

**enqueue(value);**

**break;**

**case 2:**

**dequeue();**

**break;**

**case 3:**

**display();**

**break;**

**case 4:**

**exit(0);**

**default:**

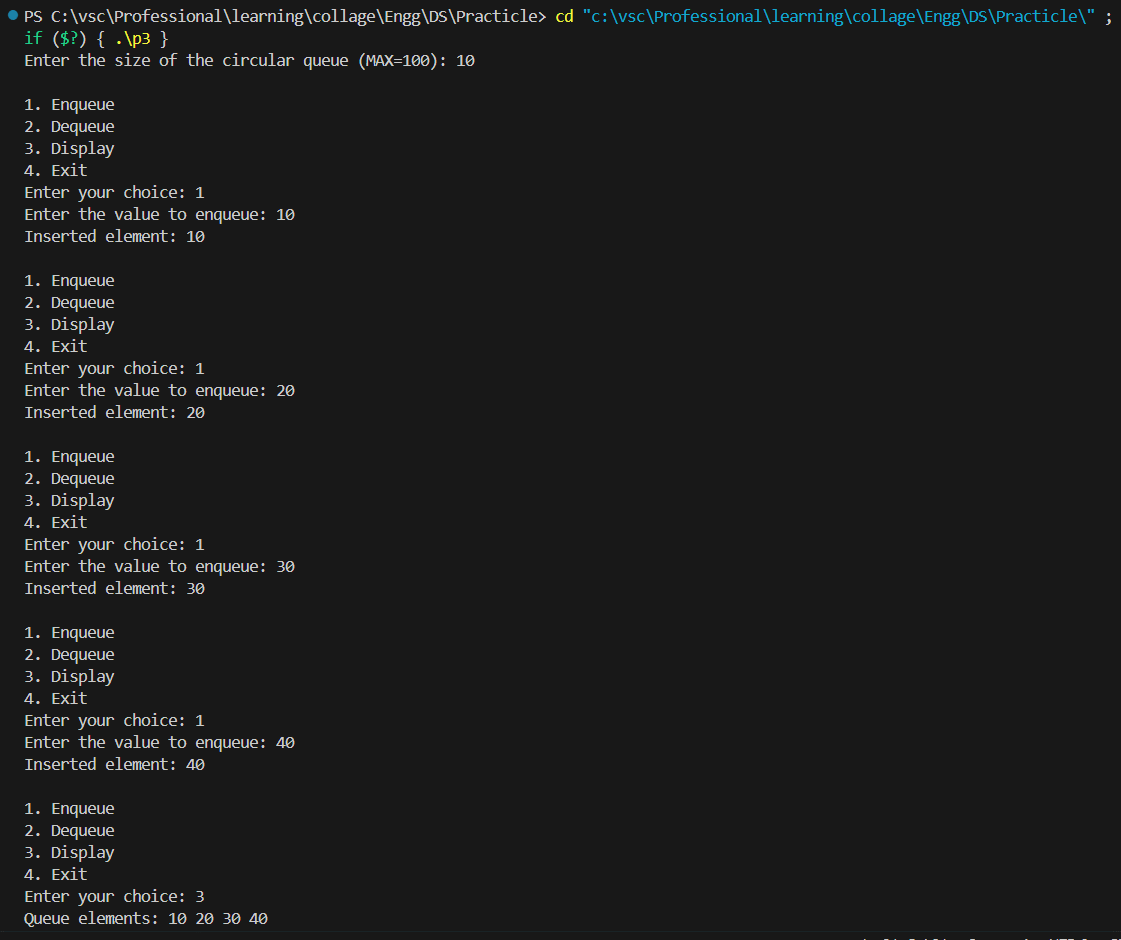
**printf("Invalid choice\n");**

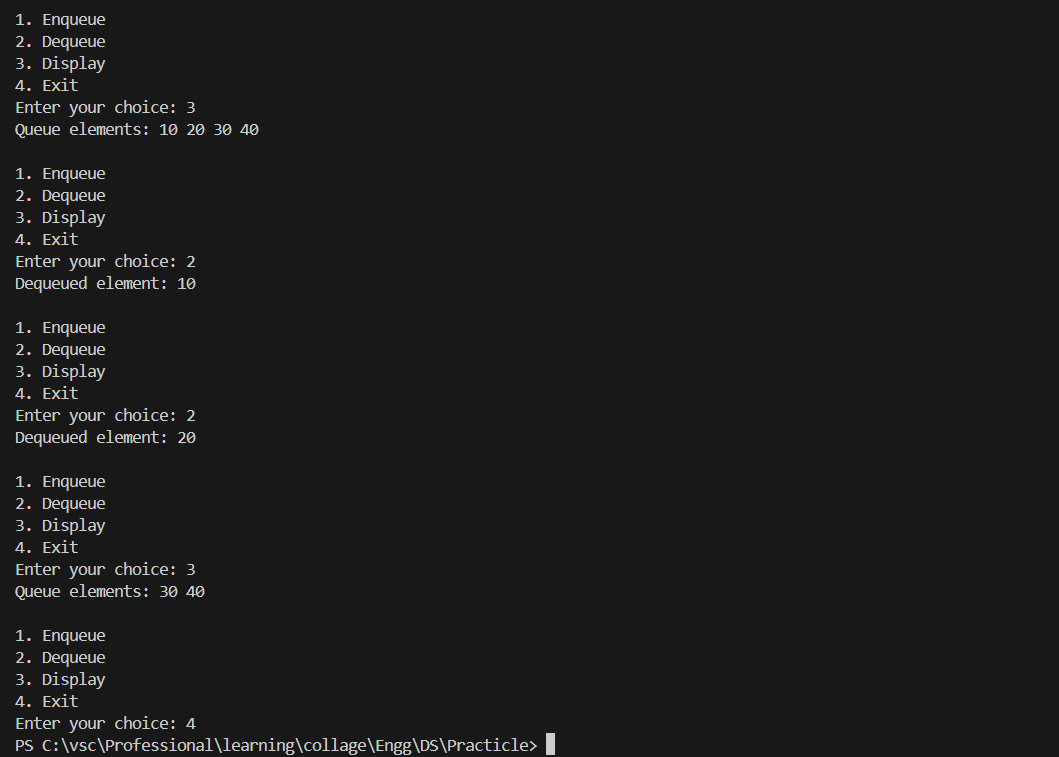
**}**

**}**

**return 0;**

**}**

**Output:**



**To implement various operations on linked list: (a)insert**

**(b)delete**

**(c) display**

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in the linked list

typedef struct Node {

int data;

struct Node\* next;

} Node;

// Function to create a new node

Node\* newNode(int data) {

Node\* new\_node = (Node\*)malloc(sizeof(Node));

new\_node->data = data;

new\_node->next = NULL;

return new\_node;

}

// Function to insert a node at the beginning of the linked list

void insertAtBeginning(Node\*\* head\_ref, int new\_data) {

Node\* new\_node = newNode(new\_data);

new\_node->next = (\*head\_ref);

(\*head\_ref) = new\_node;

}

// Function to insert a node after a given node

void insertAfter(Node\* prev\_node, int new\_data) {

if (prev\_node == NULL) {

printf("The given previous node cannot be NULL\n");

return;

}

Node\* new\_node = newNode(new\_data);

new\_node->next = prev\_node->next;

prev\_node->next = new\_node;

}

// Function to insert a node at the end of the linked list

void insertAtEnd(Node\*\* head\_ref, int new\_data) {

Node\* new\_node = newNode(new\_data);

if (\*head\_ref == NULL) {

(\*head\_ref) = new\_node;

return;

}

Node\* last = \*head\_ref;

while (last->next != NULL) {

last = last->next;

}

last->next = new\_node;

}

// Function to delete a node by its given key (value)

void deleteNode(Node\*\* head\_ref, int key) {

// Handle empty list

if (\*head\_ref == NULL) return;

// Store the head node

Node\* temp = \*head\_ref;

// If head node itself holds the key to be deleted

if (temp != NULL && temp->data == key) {

\*head\_ref = temp->next;

free(temp);

return;

}

// Search for the key to be deleted

while (temp != NULL && temp->next != NULL) {

if (temp->next->data == key) {

// Found the key. Now delete the next node

Node\* next = temp->next->next;

free(temp->next);

temp->next = next;

return;

}

temp = temp->next;

}

// If key was not present in linked list

printf("Key %d not found in linked list\n", key);

}

// Function to traverse the linked list and print its contents

void printList(Node\* node) {

while (node != NULL) {

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

node = node->next;

}

printf("\n");

}

// Function to check if the linked list is empty

int isEmpty(Node\* head) {

return head == NULL;

}

int main() {

Node\* head = NULL;

// Inserting elements at the beginning

insertAtBeginning(&head, 12);

insertAtBeginning(&head, 6);

insertAtBeginning(&head, 1);

printf("Linked list before insertion: ");

printList(head);

// Inserting a node after the node with value 6

insertAfter(head->next, 5);

printf("Linked list after insertion after 6: ");

printList(head);

// Inserting a node at the end

insertAtEnd(&head, 9);

printf("Linked list after insertion at end: ");

printList(head);

// Deleting node with value 12

deleteNode(&head, 12);

printf("Linked list after deletion of 12: ");

printList(head);

// Check if the list is empty

if (isEmpty(head)) {

printf("Linked list is empty.\n");

} else {

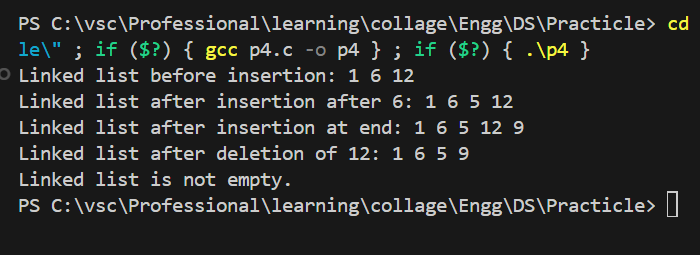
printf("Linked list is not empty.\n");

}

return 0;

}

**OUTPUT:**

****

**5. To implement stack using linked list.**

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in the linked list

typedef struct Node {

int data;

struct Node\* next;

} Node;

// Function to create a new node

Node\* newNode(int data) {

Node\* new\_node = (Node\*)malloc(sizeof(Node));

new\_node->data = data;

new\_node->next = NULL;

return new\_node;

}

// Function representing an empty stack

Node\* EmptyStack() {

return NULL;

}

// Function to push an element onto the stack

Node\* Push(Node\* stack, int data) {

Node\* new\_node = newNode(data);

new\_node->next = stack;

return new\_node;

}

// Function to pop an element from the stack

Node\* Pop(Node\* stack) {

if (stack == NULL) {

printf("Stack underflow\n");

return NULL; // Indicate error or empty stack

}

Node\* temp = stack;

int popped\_data = temp->data;

stack = stack->next;

free(temp);

return stack; // Return the updated stack

}

// Function to peek at the top element of the stack

int Peek(Node\* stack) {

if (stack == NULL) {

printf("Stack underflow\n");

return -1; // Indicate error or empty stack

}

return stack->data;

}

int main() {

Node\* stack = EmptyStack(); // Start with an empty stack

stack = Push(stack, 10);

stack = Push(stack, 20);

stack = Push(stack, 30);

printf("Top element: %d\n", Peek(stack));

stack = Pop(stack);

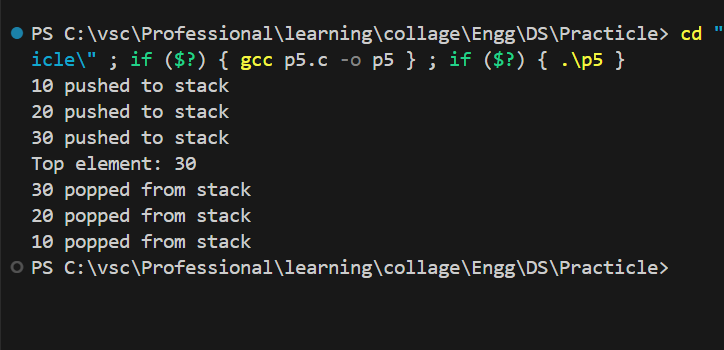
stack = Pop(stack);

stack = Pop(stack); // Try to pop from empty stack (handled in Pop function)

return 0;

}

**Output:**



**6. To implement queue using linked list.**

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in the linked list

typedef struct Node {

int data;

struct Node\* next;

} Node;

// Function to create a new node

Node\* newNode(int data) {

Node\* new\_node = (Node\*)malloc(sizeof(Node));

new\_node->data = data;

new\_node->next = NULL;

return new\_node;

}

// Structure to represent the queue

typedef struct Queue {

Node\* front; // Pointer to the front node (first element to be dequeued)

Node\* rear; // Pointer to the rear node (last element enqueued)

} Queue;

// Function to create a new queue

Queue\* createQueue() {

Queue\* queue = (Queue\*)malloc(sizeof(Queue));

queue->front = queue->rear = NULL;

return queue;

}

// Function to check if the queue is empty

int isEmpty(Queue\* queue) {

return queue->front == NULL;

}

// Function to enqueue (insert) an element into the queue

void enqueue(Queue\* queue, int data) {

Node\* new\_node = newNode(data);

if (isEmpty(queue)) { // Handle the first element being enqueued

queue->front = queue->rear = new\_node;

} else {

queue->rear->next = new\_node;

queue->rear = new\_node;

}

printf("%d enqueued to queue\n", data);

}

// Function to dequeue (remove) an element from the queue

int dequeue(Queue\* queue) {

if (isEmpty(queue)) {

printf("Queue is empty\n");

return -1; // Indicate error or empty queue

}

Node\* temp = queue->front;

int dequeued\_data = temp->data;

queue->front = queue->front->next;

if (queue->front == NULL) { // Queue becomes empty after dequeue

queue->rear = NULL;

}

free(temp);

printf("%d dequeued from queue\n", dequeued\_data);

return dequeued\_data;

}

int main() {

Queue\* queue = createQueue();

enqueue(queue, 10);

enqueue(queue, 20);

enqueue(queue, 30);

printf("Dequeued element: %d\n", dequeue(queue));

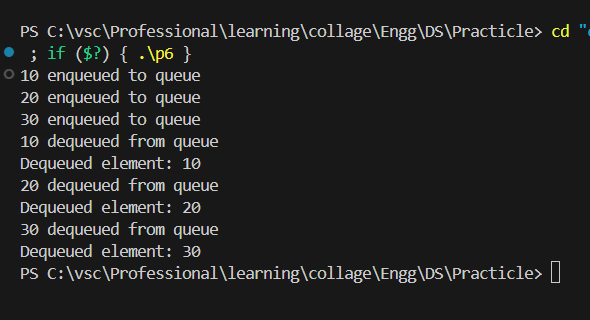
printf("Dequeued element: %d\n", dequeue(queue));

printf("Dequeued element: %d\n", dequeue(queue)); // Try to dequeue from empty queue (handled in dequeue function)

return 0;

}

**OUTPUT:**



**7. To implement linear search.**

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in the linked list

typedef struct Node {

int data;

struct Node\* next;

} Node;

// Function to create a new node

Node\* newNode(int data) {

Node\* new\_node = (Node\*)malloc(sizeof(Node));

new\_node->data = data;

new\_node->next = NULL;

return new\_node;

}

// Function to insert a node at the end of the linked list

void insertAtEnd(Node\*\* head\_ref, int new\_data) {

Node\* new\_node = newNode(new\_data);

if (\*head\_ref == NULL) {

(\*head\_ref) = new\_node;

return;

}

Node\* last = \*head\_ref;

while (last->next != NULL) {

last = last->next;

}

last->next = new\_node;

}

// Function to search for a given key (value) in the linked list

int search(Node\* head, int key) {

Node\* current = head; // Start from the beginning

int position = 1; // Track the position of the node

// Traverse the linked list

while (current != NULL) {

if (current->data == key) {

return position; // Key found, return its position

}

current = current->next;

position++;

}

return -1; // Key not found

}

int main() {

Node\* head = NULL; // Initially an empty list

int num\_elements, data;

// Get the number of elements to insert from the user

printf("Enter the number of elements in the linked list: ");

scanf("%d", &num\_elements);

// Insert elements into the linked list based on user input

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

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

scanf("%d", &data);

insertAtEnd(&head, data);

}

printf("Created linked list: ");

// Print the linked list for clarity (optional)

Node\* temp = head;

while (temp != NULL) {

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

temp = temp->next;

}

printf("\n");

// Get the key to search for from the user

int key;

printf("Enter the key to search for: ");

scanf("%d", &key);

int position = search(head, key);

if (position == -1) {

printf("Key %d not found in linked list\n", key);

} else {

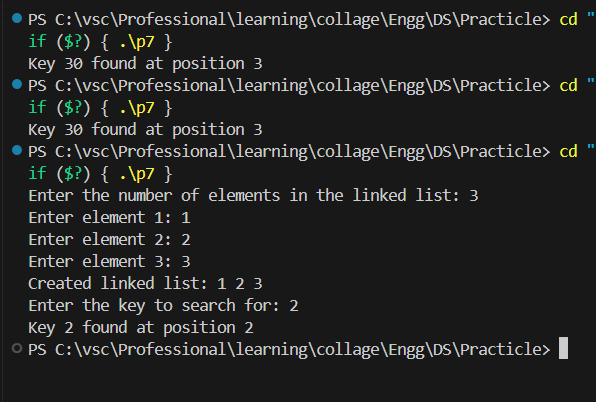
printf("Key %d found at position %d\n", key, position);

}

return 0;

}

**Output:**



**8. To implement binary search.**

#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in the linked list

typedef struct Node {

int data;

struct Node\* next;

} Node;

// Function to create a new node

Node\* newNode(int data) {

Node\* new\_node = (Node\*)malloc(sizeof(Node));

new\_node->data = data;

new\_node->next = NULL;

return new\_node;

}

// Function to insert a node in a sorted linked list (ascending order)

void insertSorted(Node\*\* head\_ref, int new\_data) {

Node\* new\_node = newNode(new\_data);

Node\* current;

/\* 1. Special case for the empty list \*/

if (\*head\_ref == NULL) {

\*head\_ref = new\_node;

return;

}

/\* 2. Traverse the list to find the insertion position \*/

current = \*head\_ref;

while (current->next != NULL && current->next->data < new\_data) {

current = current->next;

}

/\* 3. Insert node at the found position \*/

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

current->next = new\_node;

}

// Function to get the count of nodes in the linked list

int getLength(Node\* head) {

int count = 0;

Node\* current = head;

while (current != NULL) {

count++;

current = current->next;

}

return count;

}

// Function to search for a given key (value) in the linked list using binary search

int binarySearch(Node\* head, int key) {

int left = 0;

int right = getLength(head) - 1; // Adjust for 0-based indexing

// Handle empty list

if (head == NULL) {

return -1;

}

while (left <= right) {

int mid = left + (right - left) / 2;

Node\* current = head;

// Traverse to the mid-point node

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

if (current == NULL) {

return -1; // Unexpected error (list might be corrupted)

}

current = current->next;

}

// Check if key is present at mid

if (current != NULL && current->data == key) {

return mid + 1; // Return position (1-based indexing)

}

// If key is greater, ignore left half

if (current != NULL && current->data < key) {

left = mid + 1;

} else { // key is smaller, ignore right half

right = mid - 1;

}

}

// Key not found

return -1;

}

int main() {

Node\* head = NULL; // Initially an empty list

int num\_elements, data;

// Get the number of elements to insert from the user

printf("Enter the number of elements in the linked list (sorted ascending): ");

scanf("%d", &num\_elements);

// Insert elements into the linked list based on user input

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

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

scanf("%d", &data);

insertSorted(&head, data);

}

printf("Created linked list (sorted): ");

// Print the linked list for clarity (optional)

Node\* temp = head;

while (temp != NULL) {

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

temp = temp->next;

}

printf("\n");

// Get the key to search for from the user

int key;

printf("Enter the key to search for: ");

scanf("%d", &key);

int position = binarySearch(head, key);

if (position == -1) {

printf("Key %d not found in linked list\n", key);

} else {

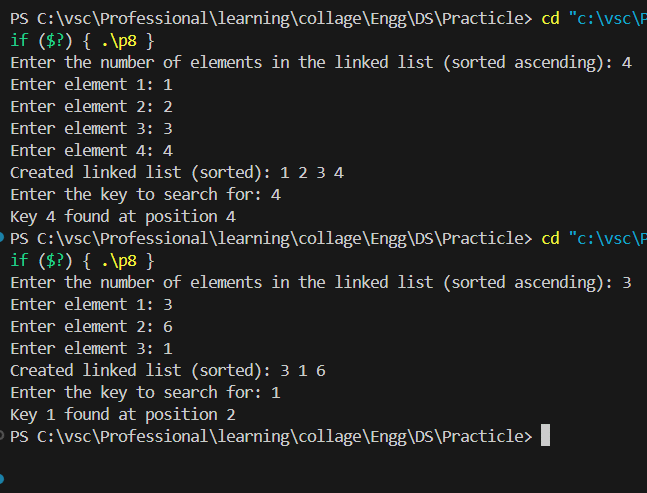
printf("Key %d found at position %d\n", key, position);

}

return 0;

}

**OUTPUT:**



**9. To implement bubble sort.**

#include <stdio.h>

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

int swapped = 1; // Flag to track if any swaps occurred in a pass

int temp;

for (int i = 0; i < n - 1 && swapped; i++) {

swapped = 0; // Reset flag for the next pass

for (int j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

// Swap elements if they are in the wrong order

temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

swapped = 1; // Set flag if a swap occurred

}

}

}

}

int main() {

int arr[100]; // Array to hold elements (adjust size as needed)

int n;

printf("Enter the number of elements (less than 100): ");

scanf("%d", &n);

printf("Enter %d elements:\n", n);

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

scanf("%d", &arr[i]);

}

printf("Unsorted array: ");

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

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

}

printf("\n");

bubbleSort(arr, n);

printf("Sorted array: ");

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

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

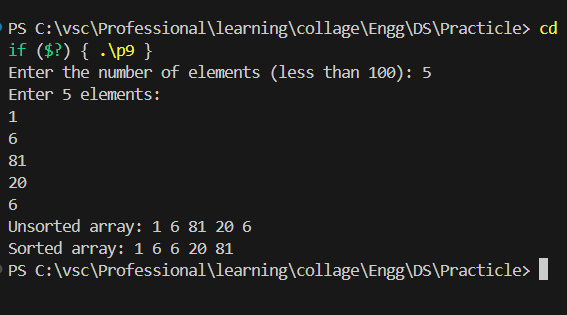
}

printf("\n");

return 0;

}

**Output:**



**10. To implement insertion sort.**

#include <stdio.h>

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

int key, j;

for (int i = 1; i < n; i++) {

key = arr[i]; // Store the current element

j = i - 1; // Initialize index for comparison

// 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--;

}

arr[j + 1] = key; // Insert the key at its correct position

}

}

int main() {

int arr[100]; // Array to hold elements (adjust size as needed)

int n;

printf("Enter the number of elements (less than 100): ");

scanf("%d", &n);

printf("Enter %d elements:\n", n);

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

scanf("%d", &arr[i]);

}

printf("Unsorted array: ");

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

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

}

printf("\n");

insertionSort(arr, n);

printf("Sorted array: ");

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

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

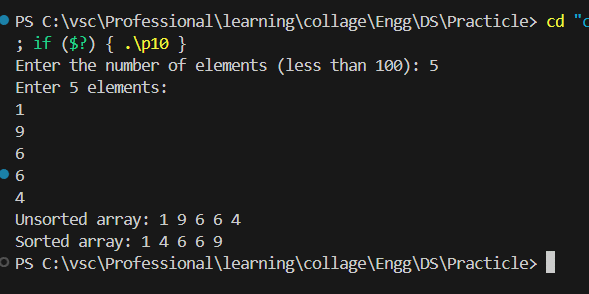
}

printf("\n");

return 0;

}

**Output:**



**11. To implement merge sort.**

#include <stdio.h>

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

  int key, j;

  for (int i = 1; i < n; i++) {

    key = arr[i];  // Store the current element

    j = i - 1;      // Initialize index for comparison

    // 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--;

    }

    arr[j + 1] = key;  // Insert the key at its correct position

  }

}

int main() {

  int arr[100]; // Array to hold elements (adjust size as needed)

  int n;

  printf("Enter the number of elements (less than 100): ");

  scanf("%d", &n);

  printf("Enter %d elements:\n", n);

  for (int i = 0; i < n; i++) {

    scanf("%d", &arr[i]);

  }

  printf("Unsorted array: ");

  for (int i = 0; i < n; i++) {

    printf("%d ", arr[i]);

  }

  printf("\n");

  insertionSort(arr, n);

  printf("Sorted array: ");

  for (int i = 0; i < n; i++) {

    printf("%d ", arr[i]);

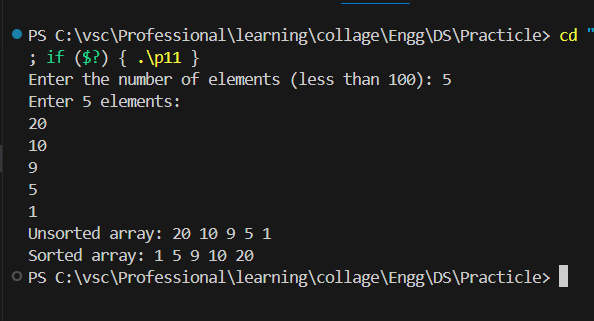
  }

  printf("\n");

  return 0;

}

**OUTPUT:**



**12. To implement quick sort**

#include <stdio.h>

// Function to swap two elements

void swap(int\* a, int\* b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Partition function for quicksort

int partition(int arr[], int low, int high) {

int pivot = arr[high]; // Select the last element as pivot

int i = (low - 1); // Index of smaller element

for (int j = low; j <= high - 1; j++) {

// Check if current element is smaller than the pivot

if (arr[j] < pivot) {

i++; // Increment index of smaller element

swap(&arr[i], &arr[j]); // Swap current element with the smaller element

}

}

swap(&arr[i + 1], &arr[high]); // Swap the pivot element with the element at i+1

return (i + 1);

}

// Recursive quicksort function

void quickSort(int arr[], int low, int high) {

if (low < high) {

// pi is partitioning index, arr[p] is now at right place

int pi = partition(arr, low, high);

// Recursively sort elements before and after partition

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

int main() {

int arr[100]; // Array to hold elements (adjust size as needed)

int n;

printf("Enter the number of elements (less than 100): ");

scanf("%d", &n);

printf("Enter %d elements:\n", n);

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

scanf("%d", &arr[i]);

}

printf("Unsorted array: ");

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

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

}

printf("\n");

quickSort(arr, 0, n - 1);

printf("Sorted array: ");

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

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

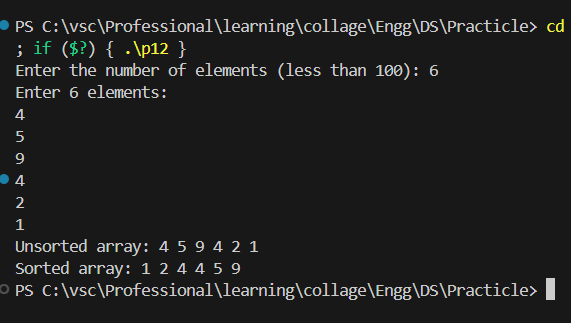
}

printf("\n");

return 0;

}

OUTPUT:



**13. Program to find the factorial of a number using recursion.**

#include <stdio.h>

// Function to calculate factorial

int factorial(int n) {

if (n == 0) { // Base case: Factorial of 0 is 1

return 1;

} else {

// Recursive case: Factorial of n is n \* factorial(n-1)

return n \* factorial(n - 1);

}

}

int main() {

int num;

printf("Enter a non-negative integer: ");

scanf("%d", &num);

if (num < 0) {

printf("Error: Factorial is not defined for negative numbers.\n");

} else {

int result = factorial(num);

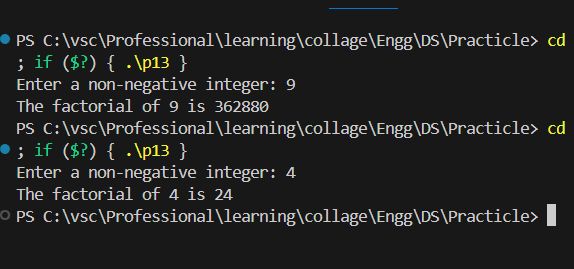
printf("The factorial of %d is %d\n", num, result);

}

return 0;

}

**OUTPUT:**



**14. To implement Heap sort.**

#include <stdio.h>

// Function to swap two elements

void swap(int\* a, int\* b) {

  int temp = \*a;

  \*a = \*b;

  \*b = temp;

}

// Function to heapify a subtree rooted with node i which is

// assumed to be a max heap

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

  int largest = i; // Initialize largest as root

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

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

  // If left child is larger than root

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

    largest = left;

  // If right child is larger than largest so far

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

    largest = right;

  // If largest is not root

  if (largest != i) {

    swap(&arr[largest], &arr[i]);

    // Recursively heapify the affected sub-tree

    heapify(arr, n, largest);

  }

}

// Function to build a max heap from the array

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

  // Start from the last parent (assuming the array is 0-indexed)

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

    heapify(arr, n, i);

}

// Function to implement Heap Sort

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

  // Build a max heap

  buildHeap(arr, n);

  // One by one extract an element from heap

  for (int i = n - 1; i > 0; i--) {

    // Move current root to end

    swap(&arr[0], &arr[i]);

    // Call max heapify on the reduced heap

    heapify(arr, i, 0);

  }

}

// Function to print the array

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

  for (int i = 0; i < n; ++i)

    printf("%d ", arr[i]);

  printf("\n");

}

int main() {

  int arr[100]; // Array to hold elements (adjust size as needed)

  int n;

  printf("Enter the number of elements (less than 100): ");

  scanf("%d", &n);

  printf("Enter %d elements:\n", n);

  for (int i = 0; i < n; i++) {

    scanf("%d", &arr[i]);

  }

  printf("Unsorted array: ");

  printArray(arr, n);

  heapSort(arr, n);

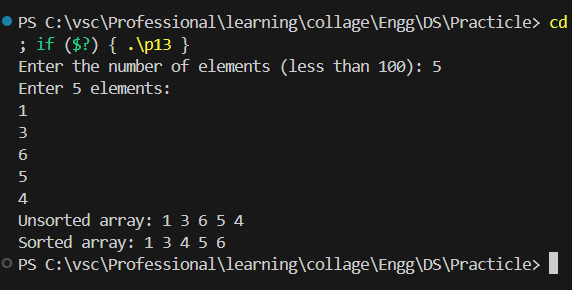
  printf("Sorted array: ");

  printArray(arr, n);

  return 0;

}

OUTPUT:



**15. Implementation of graph menu driven program.**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_VERTICES 100

typedef struct Node {

int data;

struct Node\* next;

} Node;

typedef struct Graph {

int numVertices;

Node\* adjLists[MAX\_VERTICES];

} Graph;

// Function to create a new node

Node\* newNode(int data) {

Node\* new\_node = (Node\*)malloc(sizeof(Node));

new\_node->data = data;

new\_node->next = NULL;

return new\_node;

}

// Function to create a graph with numVertices vertices

Graph\* createGraph(int numVertices) {

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

graph->numVertices = numVertices;

// Create an adjacency list for each vertex

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

graph->adjLists[i] = NULL;

}

return graph;

}

// Function to add an edge to an undirected graph

void addEdge(Graph\* graph, int src, int dest) {

// Add an edge from src to dest

Node\* new\_node = newNode(dest);

new\_node->next = graph->adjLists[src];

graph->adjLists[src] = new\_node;

// Since graph is undirected, add an edge from dest to src also

new\_node = newNode(src);

new\_node->next = graph->adjLists[dest];

graph->adjLists[dest] = new\_node;

}

// Function to print the graph

void printGraph(Graph\* graph) {

int v;

for (v = 0; v < graph->numVertices; ++v) {

printf("\n Adjacency list of vertex %d\n head ", v);

Node\* temp = graph->adjLists[v];

while (temp) {

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

temp = temp->next;

}

printf("\n");

}

}

int main() {

int numVertices, choice, src, dest;

Graph\* graph;

printf("Enter the number of vertices: ");

scanf("%d", &numVertices);

graph = createGraph(numVertices);

while (1) {

printf("\n1.Add Edge\n");

printf("2.Print the graph\n");

printf("3.Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter source vertex: ");

scanf("%d", &src);

printf("Enter destination vertex: ");

scanf("%d", &dest);

// Check if vertices are within bounds

if (src >= numVertices || dest >= numVertices) {

printf("Invalid vertex\n");

} else {

addEdge(graph, src, dest);

printf("Edge added successfully\n");

}

break;

case 2:

printGraph(graph);

break;

case 3:

printf("Exiting program...\n");

exit(0);

default:

printf("Invalid choice\n");

}

}

return 0;

}

**OUTPUT:**

