**SWASTIK COLLEGE**

Tribhuvan University

Faculty of Humanities and Social Sciences

A red text on a black background

Description automatically generated

Bachelor of Computer Application

(BCA)

Course: Data Structure and Algorithm

Semester: 3rd

A Lab Report On:

**DSA**

**Submitted to:**

Sujan Poudel

**Submitted by:**

Name: Swosti Makaju

Roll No.: 33

**LABORATORY ACTIVITIES:**

**1.Write a program to show the stack operations.**

**INPUT :**

**#include <stdio.h>**

**int MAXSIZE = 8;**

**int stack[8];**

**int top = -1;**

**int isFull() {**

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

**return 1;**

**else**

**return 0;**

**}**

**int isEmpty() {**

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

**return 1;**

**else**

**return 0;**

**}**

**void push(int data) {**

**if (!isFull()) {**

**top = top + 1;**

**stack[top] = data;**

**} else {**

**printf("Could not insert data, Stack is full.\n");**

**} }**

**int pop() {**

**int data;**

**if (!isEmpty()) {**

**data = stack[top];**

**top = top - 1;**

**return data;**

**} else {**

**printf("Could not retrieve data, Stack is empty.\n"); }**

**}**

**int peek() {**

**return stack[top];**

**}**

**void display() {**

**int i;**

**if (!isEmpty()) {**

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

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

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

**} else {**

**printf("Stack is empty.\n"); }**

**}**

**void search(int element) {**

**printf("-------------------Search Operation------------------- \n");**

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

**if (stack[i] == element) {**

**printf("Element found at index %d \n", i);**

**return;**

**}**

**}**

**printf("Element not found \n");**

**}**

**int main() {**

**push(3);**

**push(5);**

**push(9);**

**push(1);**

**push(12);**

**push(15);**

**push(12);**

**push(15);**

**push(15);**

**printf("Element at top of the stack: %d\n", peek());**

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

**display();**

**printf("\nElement popped: %d\n", pop());**

**printf("Element at top of the stack: %d\n", peek());**

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

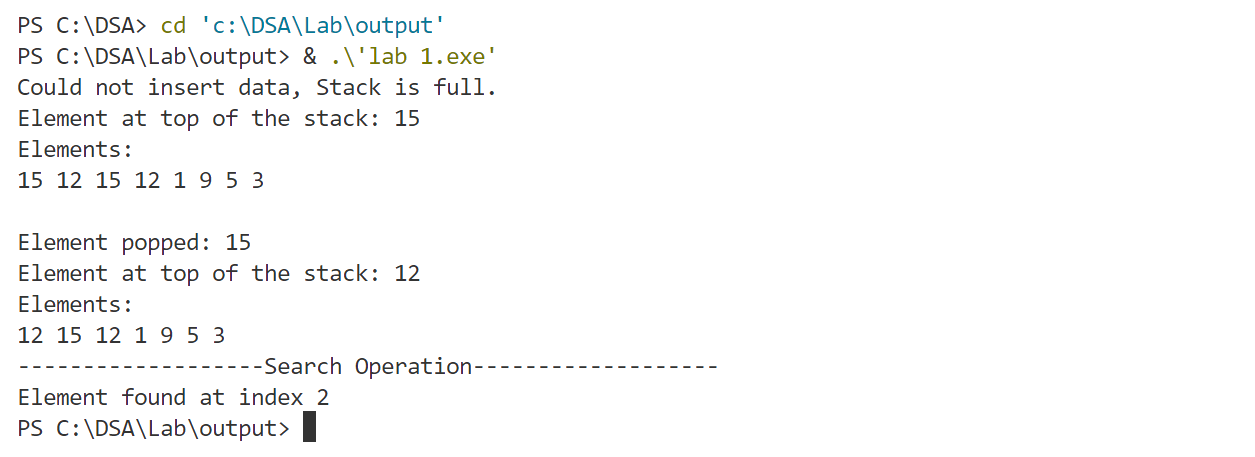
**display();**

**search(9);**

**return 0;**

**}**

**OUTPUT :**

****

**2.Write a program to implement the linear queue operations.**

**INPUT :**

**#include <stdio.h>**

**#define SIZE 5**

**void enQueue(int);**

**void deQueue();**

**void display();**

**int items[SIZE], front = -1, rear = -1;**

**int main() {**

**deQueue();**

**enQueue(1);**

**enQueue(2);**

**enQueue(3);**

**enQueue(4);**

**enQueue(5);**

**enQueue(6);**

**display();**

**display();**

**return 0;**

**}**

**void enQueue(int value) {**

**if (rear == SIZE - 1)**

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

**else {**

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

**front = 0;**

**rear++;**

**items[rear] = value;**

**printf("\nInserted -> %d", value);**

**}**

**}**

**void deQueue() {**

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

**printf("\nQueue is Empty!!");**

**else {**

**printf("\nDeleted : %d", items[front]);**

**front++;**

**if (front > rear)**

**front = rear = -1;**

**}**

**}**

**void display() {**

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

**printf("\nQueue is Empty!!!");**

**else {**

**int i;**

**printf("\nQueue elements are:\n");**

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

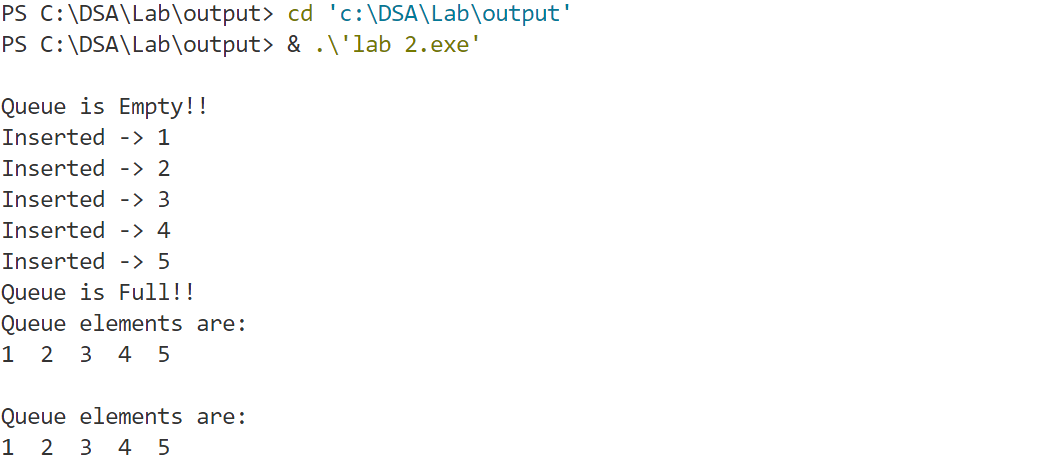
**printf("%d ", items[i]);**

**}**

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

**}**

**OUTPUT :**

****

**3.Write a program to implement circular queue.**

**INPUT :**

**#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);**

**}**

**}**

**void display() {**

**int i;**

**if (isEmpty())**

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

**else {**

**printf("\n Front -> %d ", front);**

**printf("\n Items -> ");**

**for (i = front; i != rear; i = (i + 1) % SIZE) {**

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

**}**

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

**printf("\n Rear -> %d \n", rear);**

**}**

**}**

**int main() {**

**deQueue();**

**enQueue(1);**

**enQueue(2);**

**enQueue(3);**

**enQueue(4);**

**enQueue(5);**

**enQueue(6);**

**display();**

**deQueue();**

**display();**

**enQueue(7);**

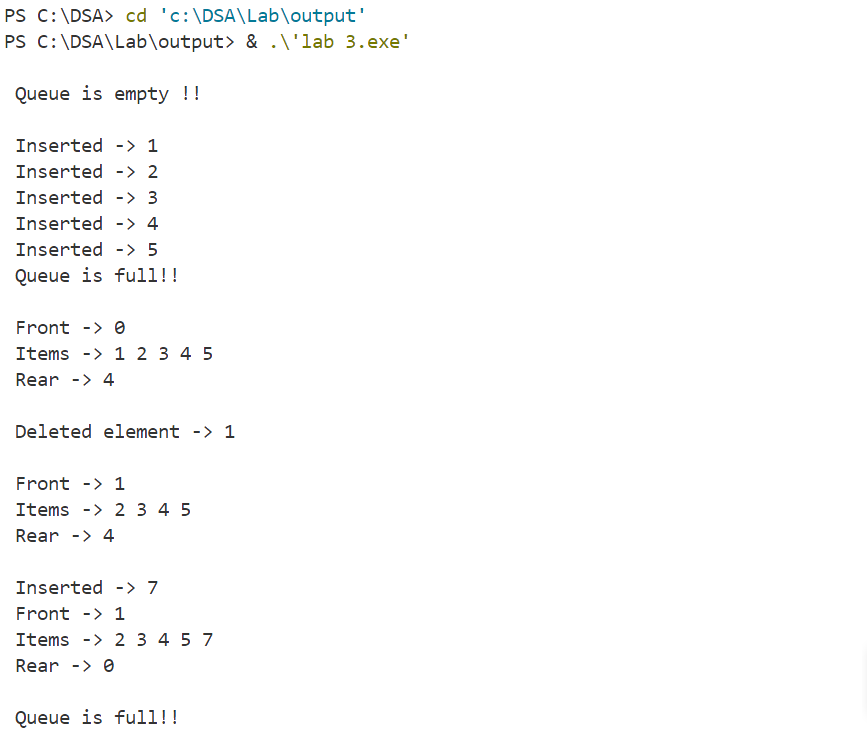
**display();**

**enQueue(8);**

**return 0;**

**}**

**OUTPUT :**

****

**4.Write a program to calculate factorial number using recursion.**

**INPUT :**

**#include<stdio.h>**

**long int multiplyNumbers(int n);**

**int main()**

**{**

**int n;**

**printf("Enter a positive integer:");**

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

**printf("Factorial of %d=%ld\n",n,multiplyNumbers(n));**

**return 0;**

**}**

**long int multiplyNumbers(int n){**

**if(n>=1)**

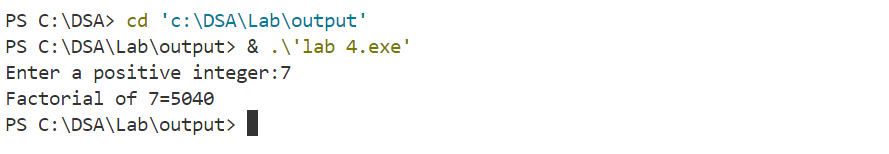
**return n\*multiplyNumbers(n-1);**

**else**

**return 1;**

**}**

**OUTPUT :**

****

**5.Write a program to check prime number using recursion.**

**INPUT :**

**#include <stdio.h>**

**int is\_prime(int n, int i) {**

**if (n <= 2)**

**return (n == 2);**

**if (n % i == 0)**

**return 0;**

**if (i \* i > n)**

**return 1;**

**return is\_prime(n, i + 1);**

**}**

**int main() {**

**int num;**

**printf("Enter a number to check if it's prime: ");**

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

**if (is\_prime(num, 2))**

**printf("%d is a prime number\n", num);**

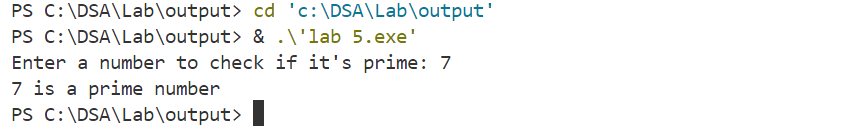
**else**

**printf("%d is not a prime number\n", num);**

**return 0;**

**}**

**OUTPUT :**

****

**6.Write a program to reverse integer number using recursion.**

**INPUT :**

**#include <stdio.h>**

**int reverse(int num) {**

**static int reversedNum = 0;**

**if (num == 0)**

**return 0;**

**int lastDigit = num % 10;**

**reversedNum = reversedNum \* 10 + lastDigit;**

**reverse(num / 10);**

**return reversedNum;**

**}**

**int main() {**

**int num;**

**printf("Enter an integer number to reverse: ");**

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

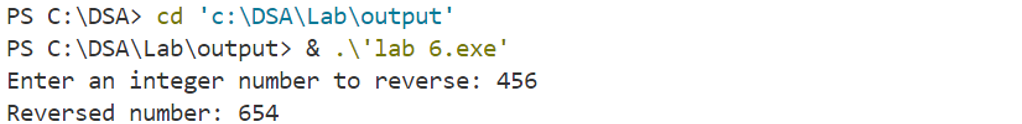
**int reversed = reverse(num);**

**printf("Reversed number: %d\n", reversed);**

**return 0;**

**}**

**OUTPUT :**

****

**7.Write a program to print Fibonacci series upto n number using recursion.**

**INPUT :**

**#include <stdio.h>**

**int fibonacci(int n) {**

**if (n <= 0) {**

**return 0;**

**} else if (n == 1) {**

**return 1;**

**} else {**

**return fibonacci(n - 1) + fibonacci(n - 2);**

**} }**

**void printFibonacciSeries(int n) {**

**int i;**

**printf("Fibonacci series up to %d terms:\n", n);**

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

**printf("%d ", fibonacci(i));**

**}**

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

**}**

**int main() {**

**int n;**

**printf("Enter the number of terms: ");**

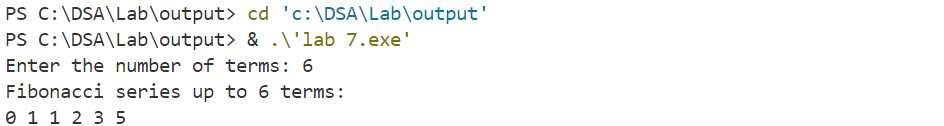
**scanf("%d", &n);**

**printFibonacciSeries(n);**

**return 0;**

**}**

**OUTPUT :**

****

**8.Write a program to solve TOH problem using recursion.**

**INPUT :**

**#include<stdio.h>**

**void hanoi(int n,char from,char to,char via){**

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

**printf("Move disk 1 from %c to %c\n",from,to);**

**}**

**else{**

**hanoi(n-1,from,via,to);**

**printf("Move disk %d from %c to %c\n",n,from,to);**

**hanoi(n-1,via,to,from);**

**}**

**}**

**int main(){**

**int n=3;**

**char from='A';**

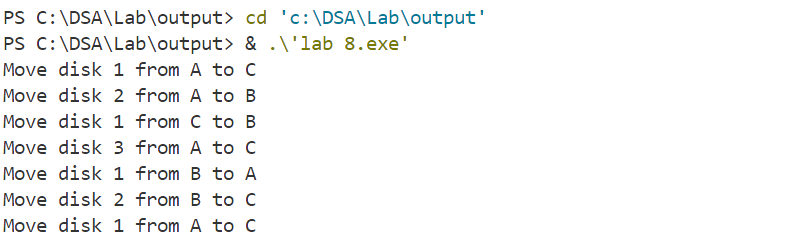
**char to='B';**

**char via='C';**

**hanoi(n,from,via,to);**

**}**

**OUTPUT :**

****

**9.Write a program to count the nodes in Linked List.**

**INPUT :**

**#include <stdio.h>**

**#include <stdlib.h>**

**struct Node {**

**int data;**

**struct Node\* next;**

**};**

**int countNodes(struct Node\* head) {**

**int count = 0;**

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

**while (current != NULL) {**

**count++;**

**current = current->next;**

**}**

**return count;**

**}**

**int main() {**

**struct Node\* head = NULL;**

**struct Node\* second = NULL;**

**struct Node\* third = NULL;**

**head = (struct Node\*)malloc(sizeof(struct Node));**

**second = (struct Node\*)malloc(sizeof(struct Node));**

**third = (struct Node\*)malloc(sizeof(struct Node));**

**head->data = 1;**

**head->next = second;**

**second->data = 2;**

**second->next = third;**

**third->data = 3;**

**third->next = NULL;**

**int node\_count = countNodes(head);**

**printf("Number of nodes in the linked list: %d\n", node\_count);**

**free(head);**

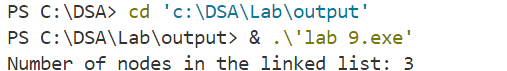
**free(second);**

**free(third);**

**return 0;**

**}**

**OUTPUT :**

****

**10.Write a program to insert and delete an item at the beginning of Singly Linked List.**

**INPUT :**

**#include <stdio.h>**

**#include <stdlib.h>**

**struct Node {**

**int data;**

**struct Node\* next;**

**};**

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

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

**if (new\_node == NULL) {**

**printf("Memory allocation failed.\n");**

**return;**

**}**

**new\_node->data = new\_data;**

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

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

**}**

**void deleteAtBeginning(struct Node\*\* head\_ref) {**

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

**printf("List is empty. Nothing to delete.\n");**

**return;**

**}**

**struct Node\* temp = \*head\_ref;**

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

**free(temp);**

**}**

**void printList(struct Node\* head) {**

**struct Node\* temp = head;**

**while (temp != NULL) {**

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

**temp = temp->next;**

**}**

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

**}**

**int main() {**

**struct Node\* head = NULL;**

**insertAtBeginning(&head, 5);**

**insertAtBeginning(&head, 10);**

**insertAtBeginning(&head, 15);**

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

**printList(head);**

**deleteAtBeginning(&head);**

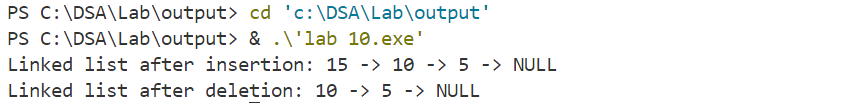
**printf("Linked list after deletion: ");**

**printList(head);**

**return 0;**

**}**

**OUTPUT :**

****

**11.Write a program to insert and delete an item at the end of Singly Linked List.**

**INPUT :**

**#include <stdio.h>**

**#include <stdlib.h>**

**struct Node {**

**int data;**

**struct Node\* next;**

**};**

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

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

**if (new\_node == NULL) {**

**printf("Memory allocation failed.\n");**

**return;**

**}**

**new\_node->data = new\_data;**

**new\_node->next = NULL;**

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

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

**return;**

**}**

**struct Node\* last = \*head\_ref;**

**while (last->next != NULL) {**

**last = last->next;**

**}   last->next = new\_node;**

**}**

**void deleteAtEnd(struct Node\*\* head\_ref) {**

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

**printf("List is empty. Nothing to delete.\n");**

**return;**

**}**

**if ((\*head\_ref)->next == NULL) {**

**free(\*head\_ref);**

**\*head\_ref = NULL;**

**return;**

**}**

**struct Node\* second\_last = \*head\_ref;**

**while (second\_last->next->next != NULL) {**

**second\_last = second\_last->next;**

**}**

**free(second\_last->next);**

**second\_last->next = NULL;**

**}**

**void printList(struct Node\* head) {**

**struct Node\* temp = head;**

**while (temp != NULL) {**

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

**temp = temp->next;**

**}**

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

**}**

**int main() {**

**struct Node\* head = NULL;**

**insertAtEnd(&head, 5);**

**insertAtEnd(&head, 10);**

**insertAtEnd(&head, 15);**

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

**printList(head);**

**deleteAtEnd(&head);**

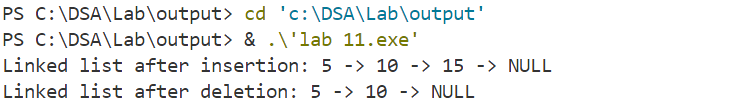
**printf("Linked list after deletion: ");**

**printList(head);**

**return 0;**

**}**

**OUTPUT :**



**12.Write a program to insert and delete an item at specified location of Singly Linked List.**

**INPUT :**

**#include <stdio.h>**

**#include <stdlib.h>**

**struct Node {**

**int data;**

**struct Node\* next;**

**};**

**void insertAtPosition(struct Node\*\* head\_ref, int new\_data, int position) {**

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

**if (new\_node == NULL) {**

**printf("Memory allocation failed.\n");**

**return;**

**}**

**new\_node->data = new\_data;**

**if (\*head\_ref == NULL || position == 1) {**

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

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

**return;**

**}**

**struct Node\* current = \*head\_ref;**

**for (int i = 1; i < position - 1 && current != NULL; i++) {**

**current = current->next;**

**}**

**if (current == NULL) {**

**printf("Position is greater than the number of nodes. Inserting at the end.\n");**

**current = \*head\_ref;**

**while (current->next != NULL) {**

**current = current->next;**

**} }**

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

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

**}**

**void deleteAtPosition(struct Node\*\* head\_ref, int position) {**

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

**printf("List is empty. Nothing to delete.\n");**

**return;**

**}**

**struct Node\* temp = \*head\_ref;**

**if (position == 1) {**

**\*head\_ref = temp->next;**

**free(temp);**

**return;**

**}**

**for (int i = 1; temp != NULL && i < position - 1; i++) {**

**temp = temp->next;**

**}**

**if (temp == NULL || temp->next == NULL) {**

**printf("Position is greater than the number of nodes.\n");**

**return;**

**}**

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

**free(temp->next);**

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

**}**

**void printList(struct Node\* head) {**

**struct Node\* temp = head;**

**while (temp != NULL) {**

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

**temp = temp->next;**

**}**

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

**}**

**int main() {**

**struct Node\* head = NULL;**

**insertAtPosition(&head, 5, 1);**

**insertAtPosition(&head, 10, 2);**

**insertAtPosition(&head, 15, 3);**

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

**printList(head);**

**deleteAtPosition(&head, 2);**

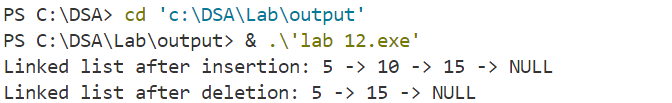
**printf("Linked list after deletion: ");**

**printList(head);**

**return 0;**

**}**

**OUTPUT :**



**13. Write a program to insert and delete an item at the beginning of Doubly Linked List.**

**INPUT :**

**#include <stdio.h>**

**#include <stdlib.h>**

**struct Node {**

**int data;**

**struct Node\* next;**

**struct Node\* prev;**

**};**

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

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

**if (new\_node == NULL) {**

**printf("Memory allocation failed.\n");**

**return;**

**}**

**new\_node->data = new\_data;**

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

**new\_node->prev = NULL;**

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

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

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

**}**

**void deleteAtBeginning(struct Node\*\* head\_ref) {**

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

**printf("List is empty. Nothing to delete.\n");**

**return;**

**}**

**struct Node\* temp = (\*head\_ref);**

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

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

**(\*head\_ref)->prev = NULL;**

**free(temp);**

**}**

**void printList(struct Node\* head) {**

**struct Node\* temp = head;**

**while (temp != NULL) {**

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

**temp = temp->next;**

**}**

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

**}**

**int main() {**

**struct Node\* head = NULL;**

**insertAtBeginning(&head, 5);**

**insertAtBeginning(&head, 10);**

**insertAtBeginning(&head, 15);**

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

**printList(head);**

**deleteAtBeginning(&head);**

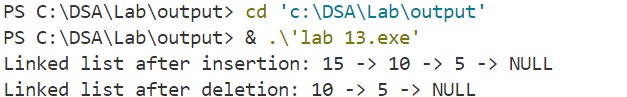
**printf("Linked list after deletion: ");**

**printList(head);**

**return 0;**

**}**

**OUTPUT :**



**14.Write a program to insert and delete an item at the end of Doubly Linked List.**

**INPUT :**

**#include <stdio.h>**

**#include <stdlib.h>**

**struct Node {**

**int data;**

**struct Node\* next;**

**struct Node\* prev;**

**};**

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

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

**if (new\_node == NULL) {**

**printf("Memory allocation failed.\n");**

**return;**

**}**

**new\_node->data = new\_data;**

**new\_node->next = NULL;**

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

**new\_node->prev = NULL;**

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

**return;**

**}**

**struct Node\* last = \*head\_ref;**

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

**last = last->next;**

**last->next = new\_node;**

**new\_node->prev = last;**

**}**

**void deleteAtEnd(struct Node\*\* head\_ref) {**

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

**printf("List is empty. Nothing to delete.\n");**

**return;**

**}**

**struct Node\* last = \*head\_ref;**

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

**last = last->next;**

**if (last->prev == NULL) {**

**\*head\_ref = NULL;**

**free(last);**

**return;**

**}**

**last->prev->next = NULL;**

**free(last);**

**}**

**void printList(struct Node\* head) {**

**struct Node\* temp = head;**

**while (temp != NULL) {**

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

**temp = temp->next;**

**}**

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

**}**

**int main() {**

**struct Node\* head = NULL;**

**insertAtEnd(&head, 5);**

**insertAtEnd(&head, 10);**

**insertAtEnd(&head, 15);**

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

**printList(head);**

**deleteAtEnd(&head);**

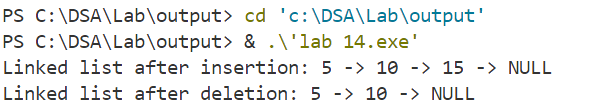
**printf("Linked list after deletion: ");**

**printList(head);**

**return 0;**

**}**

**OUTPUT :**



**15. Write a program to insert and delete an item at specified location of Doubly Linked List.**

**INPUT :**

**#include <stdio.h>**

**#include <stdlib.h>**

**struct Node {**

**int data;**

**struct Node\* next;**

**struct Node\* prev;**

**};**

**void insertAtPosition(struct Node\*\* head\_ref, int new\_data, int position) {**

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

**if (new\_node == NULL) {**

**printf("Memory allocation failed.\n");**

**return;**

**}**

**new\_node->data = new\_data;**

**if (position == 1) {**

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

**new\_node->prev = NULL;**

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

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

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

**return;**

**}**

**struct Node\* current = \*head\_ref;**

**int i;**

**for (i = 1; current != NULL && i < position - 1; i++)**

**current = current->next;**

**if (current == NULL) {**

**printf("Position is greater than the number of nodes. Inserting at the end.\n");**

**insertAtEnd(head\_ref, new\_data);**

**return;**

**}**

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

**if (current->next != NULL)**

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

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

**new\_node->prev = current;**

**}**

**void deleteAtPosition(struct Node\*\* head\_ref, int position) {**

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

**printf("List is empty. Nothing to delete.\n");**

**return;**

**}**

**struct Node\* current = \*head\_ref;**

**if (position == 1) {**

**\*head\_ref = current->next;**

**if (current->next != NULL)**

**current->next->prev = NULL;**

**free(current);**

**return;**

**}**

**int i;**

**for (i = 1; current != NULL && i < position; i++)**

**current = current->next;**

**if (current == NULL) {**

**printf("Position is greater than the number of nodes.\n");**

**return;**

**}**

**if (current->next != NULL)**

**current->next->prev = current->prev;**

**current->prev->next = current->next;**

**free(current);**

**}**

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

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

**if (new\_node == NULL) {**

**printf("Memory allocation failed.\n");**

**return;**

**}**

**new\_node->data = new\_data;**

**new\_node->next = NULL;**

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

**new\_node->prev = NULL;**

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

**return;**

**}**

**struct Node\* last = \*head\_ref;**

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

**last = last->next;**

**last->next = new\_node;**

**new\_node->prev = last;**

**}**

**void printList(struct Node\* head) {**

**struct Node\* temp = head;**

**while (temp != NULL) {**

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

**temp = temp->next;**

**}**

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

**}**

**int main() {**

**struct Node\* head = NULL;**

**insertAtEnd(&head, 5);**

**insertAtEnd(&head, 10);**

**insertAtEnd(&head, 15);**

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

**printList(head);**

**insertAtPosition(&head, 20, 2);**

**printf("Linked list after insertion at position 2: ");**

**printList(head);**

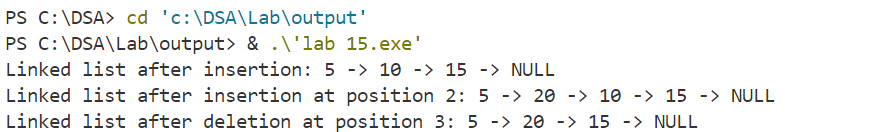
**deleteAtPosition(&head, 3);**

**printf("Linked list after deletion at position 3: ");**

**printList(head);**

**return 0;**

**}**

**OUTPUT :** 

**16.Insert and delete node from circular linked list.**

**INPUT :**

**#include <stdio.h>**

**#include <stdlib.h>**

**struct Node {**

**int data;**

**struct Node\* next;**

**};**

**struct Node\* createNode(int data) {**

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

**if (newNode == NULL) {**

**printf("Memory allocation failed.\n");**

**exit(1);**

**}**

**newNode->data = data;**

**newNode->next = NULL;**

**return newNode;**

**}**

**void insertAtBeginning(struct Node\*\* head\_ref, int data) {**

**struct Node\* newNode = createNode(data);**

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

**newNode->next = newNode;**

**\*head\_ref = newNode;**

**} else {**

**struct Node\* last = \*head\_ref;**

**while (last->next != \*head\_ref) {**

**last = last->next;**

**}**

**newNode->next = \*head\_ref;**

**last->next = newNode;**

**\*head\_ref = newNode;**

**}**

**}**

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

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

**printf("List is empty. Nothing to delete.\n");**

**return;**

**}**

**struct Node \*temp = \*head\_ref, \*prev;**

**if (temp->data == key && temp->next == \*head\_ref) {**

**free(temp);**

**\*head\_ref = NULL;**

**return;**

**}**

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

**while (temp->next != \*head\_ref)**

**temp = temp->next;**

**temp->next = (\*head\_ref)->next;**

**free(\*head\_ref);**

**\*head\_ref = temp->next;**

**}**

**while (temp->next != \*head\_ref && temp->data != key) {**

**prev = temp;**

**temp = temp->next;**

**}**

**if (temp->data != key) {**

**printf("Node with data %d not found in the list.\n", key);**

**return;**

**}**

**prev->next = temp->next;**

**free(temp);**

**}**

**void printList(struct Node\* head) {**

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

**printf("List is empty.\n");**

**return;**

**}**

**struct Node\* temp = head;**

**do {**

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

**temp = temp->next;**

**} while (temp != head);**

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

**}**

**int main() {**

**struct Node\* head = NULL;**

**insertAtBeginning(&head, 5);**

**insertAtBeginning(&head, 10);**

**insertAtBeginning(&head, 15);**

**printf("Initial circular linked list: ");**

**printList(head);**

**deleteNode(&head, 10);**

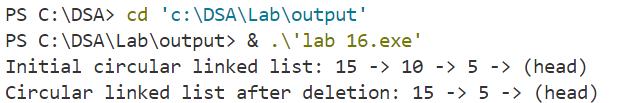
**printf("Circular linked list after deletion: ");**

**printList(head);**

**return 0;**

**}**

**OUTPUT :**



**17.Write a program to create binary search tree using given set of nodes.**

**INPUT :**

**#include <stdio.h>**

**#include <stdlib.h>**

**struct Node {**

**int data;**

**struct Node\* left;**

**struct Node\* right;**

**};**

**struct Node\* createNode(int data) {**

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

**if (newNode == NULL) {**

**printf("Memory allocation failed.\n");**

**exit(1);**

**}**

**newNode->data = data;**

**newNode->left = newNode->right = NULL;**

**return newNode;**

**}**

**struct Node\* insert(struct Node\* root, int data) {**

**if (root == NULL) {**

**return createNode(data);**

**}**

**if (data < root->data) {**

**root->left = insert(root->left, data);**

**} else if (data > root->data) {**

**root->right = insert(root->right, data);**

**}**

**return root;**

**}**

**void inorderTraversal(struct Node\* root) {**

**if (root != NULL) {**

**inorderTraversal(root->left);**

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

**inorderTraversal(root->right);**

**} }**

**int main() {**

**int nodes[] = {5, 3, 8, 2, 4, 7, 9};**

**struct Node\* root = NULL;**

**int i;**

**for (i = 0; i < sizeof(nodes) / sizeof(nodes[0]); i++) {**

**root = insert(root, nodes[i]);**

**}**

**printf("Inorder traversal of the created BST: ");**

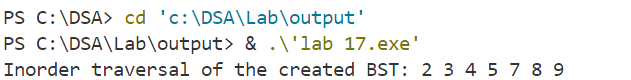
**inorderTraversal(root);**

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

**return 0;**

**}**

**OUTPUT :**



**18.Write a program to implement binary tree traversal methods(pre-order,in-order and post-order).**

**INPUT :**

**#include <stdio.h>**

**#include <stdlib.h>**

**struct Node {**

**int data;**

**struct Node\* left;**

**struct Node\* right;**

**};**

**struct Node\* createNode(int data) {**

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

**if (newNode == NULL) {**

**printf("Memory allocation failed.\n");**

**exit(1);**

**}**

**newNode->data = data;**

**newNode->left = newNode->right = NULL;**

**return newNode;**

**}**

**void preorderTraversal(struct Node\* root) {**

**if (root != NULL) {**

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

**preorderTraversal(root->left);**

**preorderTraversal(root->right);**

**}**

**}**

**void inorderTraversal(struct Node\* root) {**

**if (root != NULL) {**

**inorderTraversal(root->left);**

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

**inorderTraversal(root->right);**

**}**

**}**

**void postorderTraversal(struct Node\* root) {**

**if (root != NULL) {**

**postorderTraversal(root->left);**

**postorderTraversal(root->right);**

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

**}**

**}**

**int main() {**

**struct Node\* root = createNode(1);**

**root->left = createNode(2);**

**root->right = createNode(3);**

**root->left->left = createNode(4);**

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

**printf("Pre-order traversal: ");**

**preorderTraversal(root);**

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

**printf("In-order traversal: ");**

**inorderTraversal(root);**

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

**printf("Post-order traversal: ");**

**postorderTraversal(root);**

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

**return 0;**

**}**

**OUTPUT :**

****

**19.Write a program to construct AVL tree using given set of data.**

**INPUT :**

**#include <stdio.h>**

**#include <stdlib.h>**

**struct Node {**

**int data;**

**struct Node\* left;**

**struct Node\* right;**

**int height;**

**};**

**int height(struct Node\* node) {**

**if (node == NULL)**

**return 0;**

**return node->height;**

**}**

**int max(int a, int b) {**

**return (a > b) ? a : b;**

**}**

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

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

**if (node == NULL) {**

**printf("Memory allocation failed.\n");**

**exit(1);**

**}**

**node->data = data;**

**node->left = node->right = NULL;**

**node->height = 1;**

**return (node);**

**}**

**struct Node\* rightRotate(struct Node\* y) {**

**struct Node\* x = y->left;**

**struct Node\* T2 = x->right;**

**x->right = y;**

**y->left = T2;**

**y->height = max(height(y->left), height(y->right)) + 1;**

**x->height = max(height(x->left), height(x->right)) + 1;**

**return x;**

**}**

**struct Node\* leftRotate(struct Node\* x) {**

**struct Node\* y = x->right;**

**struct Node\* T2 = y->left;**

**y->left = x;**

**x->right = T2;**

**x->height = max(height(x->left), height(x->right)) + 1;**

**y->height = max(height(y->left), height(y->right)) + 1;**

**return y;**

**}**

**int getBalance(struct Node\* node) {**

**if (node == NULL)**

**return 0;**

**return height(node->left) - height(node->right);**

**}**

**struct Node\* insert(struct Node\* node, int data) {**

**if (node == NULL)**

**return (newNode(data));**

**if (data < node->data)**

**node->left = insert(node->left, data);**

**else if (data > node->data)**

**node->right = insert(node->right, data);**

**else**

**return node;**

**node->height = 1 + max(height(node->left), height(node->right));**

**int balance = getBalance(node);**

**if (balance > 1 && data < node->left->data)**

**return rightRotate(node);**

**if (balance < -1 && data > node->right->data)**

**return leftRotate(node);**

**if (balance > 1 && data > node->left->data) {**

**node->left = leftRotate(node->left);**

**return rightRotate(node);**

**}**

**if (balance < -1 && data < node->right->data) {**

**node->right = rightRotate(node->right);**

**return leftRotate(node);**

**}**

**return node;**

**}**

**void preOrder(struct Node\* root) {**

**if (root != NULL) {**

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

**preOrder(root->left);**

**preOrder(root->right);**

**}**

**}**

**int main() {**

**struct Node\* root = NULL;**

**int data[] = {10, 20, 30, 40, 50, 25};**

**int n = sizeof(data) / sizeof(data[0]);**

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

**root = insert(root, data[i]);**

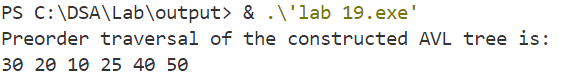
**printf("Preorder traversal of the constructed AVL tree is: \n");**

**preOrder(root);**

**return 0;**

**}**

**OUTPUT :**



**20.Write a program to sort an array using Bubble Sort.**

**INPUT :**

**#include <stdio.h>**

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

**int i, j, temp;**

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

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

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

**temp = arr[j];**

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

**arr[j+1] = temp; } } } }**

**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("Original array: \n");**

**printArray(arr, n);**

**bubbleSort(arr, n);**

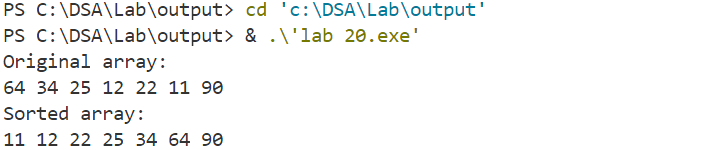
**printf("Sorted array: \n");**

**printArray(arr, n);**

**return 0;**

**}**

**OUTPUT :**

****

**21.Write a program to sort an array using Insertion Sort.**

**INPUT :**

**#include <stdio.h>**

**int main() {**

**int arr[] = {5, 4, 6, 2, 11};**

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

**printf("Unsorted array: ");**

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

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

**}**

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

**int current = arr[i];**

**int j = i - 1;**

**while (j >= 0 && current < arr[j]) {**

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

**j--;**

**}**

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

**}**

**printf("\nSorted array: ");**

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

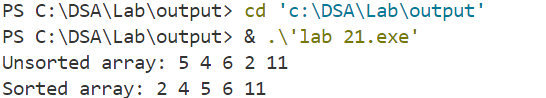
**printf("%d ", arr[i]);**

**}**

**return 0;**

**}**

**OUTPUT :**

****

**22.Write a program to sort an array using Selection Sort.**

**INPUT :**

**#include <stdio.h>**

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

**int temp = \*a;**

**\*a = \*b;**

**\*b = temp;**

**}**

**int main() {**

**int arr[] = {5, 4, 6, 2, 11};**

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

**printf("Unsorted array: ");**

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

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

**}**

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

**int min\_index = i;**

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

**if (arr[j] < arr[min\_index]) {**

**min\_index = j;**

**} }**

**int temp = arr[i];**

**arr[i] = arr[min\_index];**

**arr[min\_index] = temp;**

**}**

**printf("\nSorted array: ");**

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

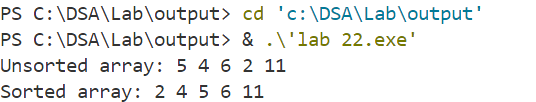
**printf("%d ", arr[i]);**

**}**

**return 0;**

**}**

**OUTPUT :**

****

**23.Write a program to sort an array using Merge Sort.**

**INPUT :**

**#include <stdio.h>**

**void merge(int arr[], int left, int middle, int right) {**

**int i, j, k;**

**int n1 = middle - left + 1;**

**int n2 = right - middle;**

**int L[n1], R[n2];**

**for (i = 0; i < n1; i++)**

**L[i] = arr[left + i];**

**for (j = 0; j < n2; j++)**

**R[j] = arr[middle + 1 + j];**

**i = 0;**

**j = 0;**

**k = left;**

**while (i < n1 && j < n2) {**

**if (L[i] <= R[j]) {**

**arr[k] = L[i];**

**i++;**

**} else {**

**arr[k] = R[j];**

**j++;**

**}**

**k++;**

**}**

**while (i < n1) {**

**arr[k] = L[i];**

**i++;**

**k++;**

**}**

**while (j < n2) {**

**arr[k] = R[j];**

**j++;**

**k++;**

**}**

**}**

**void mergeSort(int arr[], int left, int right) {**

**if (left < right) {**

**int middle = left + (right - left) / 2;**

**mergeSort(arr, left, middle);**

**mergeSort(arr, middle + 1, right);**

**merge(arr, left, middle, right);**

**}**

**}**

**int main() {**

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

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

**printf("Unsorted array: ");**

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

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

**}**

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

**mergeSort(arr, 0, n - 1);**

**printf("Sorted array: ");**

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

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

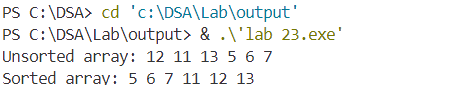
**}**

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

**return 0;**

**}**

**OUTPUT :**

****

**24.Write a program to sort an array using Quick Sort.**

**INPUT :**

**#include <stdio.h>**

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

**int temp = \*a;**

**\*a = \*b;**

**\*b = temp;**

**}**

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

**int pivot = arr[high];**

**int i = (low - 1);**

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

**if (arr[j] < pivot) {**

**i++;**

**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) {**

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

**quickSort(arr, low, pi - 1);**

**quickSort(arr, pi + 1, high);**

**}**

**}**

**int main() {**

**int arr[] = {10, 7, 8, 9, 1, 5};**

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

**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]);**

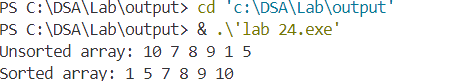
**}**

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

**return 0;**

**}**

**OUTPUT :**

****

**25.Write a program to construct a heap using given set of data.**

**INPUT :**

**#include <stdio.h>**

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

**int temp = \*a;**

**\*a = \*b;**

**\*b = temp; }**

**void heapify(int array[], int n, int i) {**

**int largest = i;**

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

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

**if (left < n && array[left] > array[largest])**

**largest = left;**

**if (right < n && array[right] > array[largest])**

**largest = right;**

**if (largest != i) {**

**swap(&array[i], &array[largest]);**

**heapify(array, n, largest);**

**}**

**}**

**void buildHeap(int array[], int n) {**

**int startIdx = (n / 2) - 1;**

**for (int i = startIdx; i >= 0; i--) {**

**heapify(array, n, i);**

**}**

**}**

**void printHeap(int array[], int n) {**

**printf("Array representation of Heap:\n");**

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

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

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

**}**

**int main() {**

**int array[] = {4, 10, 3, 5, 1};**

**int n = sizeof(array) / sizeof(array[0]);**

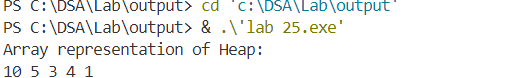
**buildHeap(array, n);**

**printHeap(array, n);**

**return 0;**

**}**

**OUTPUT :**

****

**26.Write a program to sort an array using Heap Sort.**

**INPUT :**

**#include <stdio.h>**

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

**int temp = \*a;**

**\*a = \*b;**

**\*b = temp;**

**}**

**void heapify(int array[], int n, int i) {**

**int largest = i;**

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

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

**if (left < n && array[left] > array[largest])**

**largest = left;**

**if (right < n && array[right] > array[largest])**

**largest = right;**

**if (largest != i) {**

**swap(&array[i], &array[largest]);**

**heapify(array, n, largest);**

**}**

**}**

**void buildHeap(int array[], int n) {**

**int startIdx = (n / 2) - 1;**

**for (int i = startIdx; i >= 0; i--) {**

**heapify(array, n, i);**

**}**

**}**

**void heapSort(int array[], int n) {**

**buildHeap(array, n);**

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

**swap(&array[0], &array[i]);**

**heapify(array, i, 0);**

**}**

**}**

**void printArray(int array[], int n) {**

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

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

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

**}**

**int main() {**

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

**int n = sizeof(array) / sizeof(array[0]);**

**printf("Original array:\n");**

**printArray(array, n);**

**heapSort(array, n);**

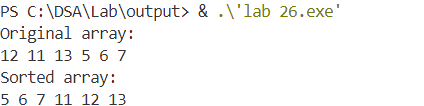
**printf("Sorted array:\n");**

**printArray(array, n);**

**return 0;**

**}**

**OUTPUT :**

****

**27.Write a program to search an item from an array using sequential search.**

**INPUT :**

**#include <stdio.h>**

**int sequentialSearch(int array[], int n, int key) {**

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

**if (array[i] == key) {**

**return i;**

**}**

**}**

**return -1;**

**}**

**void printArray(int array[], int n) {**

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

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

**}**

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

**}**

**int main() {**

**int array[] = {4, 2, 7, 1, 9, 3};**

**int n = sizeof(array) / sizeof(array[0]);**

**int key = 7;**

**printf("Array: ");**

**printArray(array, n);**

**int result = sequentialSearch(array, n, key);**

**if (result != -1) {**

**printf("Element %d found at index %d\n", key, result);**

**} else {**

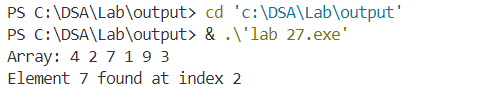
**printf("Element %d not found in the array\n", key);**

**}**

**return 0;**

**}**

**OUTPUT :**

****

**28. Write a program to search an item from an array using binary search.**

**INPUT :**

**#include <stdio.h>**

**int binarySearch(int array[], int low, int high, int key) {**

**while (low <= high) {**

**int mid = low + (high - low) / 2;**

**if (array[mid] == key)**

**return mid;**

**if (array[mid] < key)**

**low = mid + 1;**

**else**

**high = mid - 1;**

**}**

**return -1;**

**}**

**void printArray(int array[], int n) {**

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

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

**}**

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

**}**

**int main() {**

**int array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};**

**int n = sizeof(array) / sizeof(array[0]);**

**int key = 7;**

**printf("Array: ");**

**printArray(array, n);**

**int result = binarySearch(array, 0, n - 1, key);**

**if (result != -1) {**

**printf("Element %d found at index %d\n", key, result);**

**} else {**

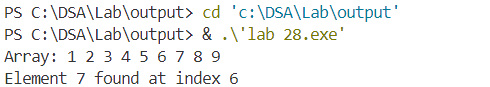
**printf("Element %d not found in the array\n", key);**

**}**

**return 0;**

**}**

**OUTPUT :**

****

**29. Write a program to represent the graph.**

**INPUT :**

**#include <stdio.h>**

**#include <stdlib.h>**

**struct AdjListNode {**

**int dest;**

**struct AdjListNode\* next;**

**};**

**struct AdjList {**

**struct AdjListNode\* head;**

**};**

**struct Graph {**

**int V;**

**struct AdjList\* array;**

**};**

**struct AdjListNode\* newAdjListNode(int dest) {**

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

**newNode->dest = dest;**

**newNode->next = NULL;**

**return newNode;**

**}**

**struct Graph\* createGraph(int V) {**

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

**graph->V = V;**

**graph->array = (struct AdjList\*) malloc(V \* sizeof(struct AdjList));**

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

**graph->array[i].head = NULL;**

**return graph;**

**}**

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

**struct AdjListNode\* newNode = newAdjListNode(dest);**

**newNode->next = graph->array[src].head;**

**graph->array[src].head = newNode;**

**newNode = newAdjListNode(src);**

**newNode->next = graph->array[dest].head;**

**graph->array[dest].head = newNode;**

**}**

**void printGraph(struct Graph\* graph) {**

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

**struct AdjListNode\* pCrawl = graph->array[v].head;**

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

**while (pCrawl) {**

**printf("-> %d", pCrawl->dest);**

**pCrawl = pCrawl->next;**

**}**

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

**}**

**}**

**int main() {**

**int V = 5;**

**struct Graph\* graph = createGraph(V);**

**addEdge(graph, 0, 1);**

**addEdge(graph, 0, 4);**

**addEdge(graph, 1, 2);**

**addEdge(graph, 1, 3);**

**addEdge(graph, 1, 4);**

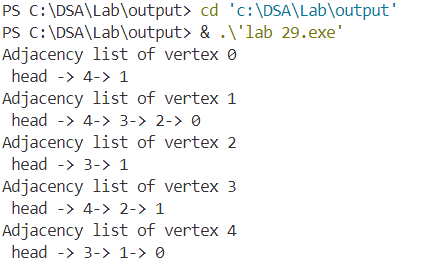
**addEdge(graph, 2, 3);**

**addEdge(graph, 3, 4);**

**printGraph(graph);**

**return 0;**

**}  
OUTPUT :**

****

**30. Write a program to traverse (BFS,DFS) in a graph.**

**INPUT :**

**#include <stdio.h>**

**#include <stdlib.h>**

**#define MAX 100**

**struct AdjListNode {**

**int dest;**

**struct AdjListNode\* next;**

**};**

**struct AdjList {**

**struct AdjListNode\* head;**

**};**

**struct Graph {**

**int V;**

**struct AdjList\* array;**

**};**

**struct AdjListNode\* newAdjListNode(int dest) {**

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

**newNode->dest = dest;**

**newNode->next = NULL;**

**return newNode;**

**}**

**struct Graph\* createGraph(int V) {**

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

**graph->V = V;**

**graph->array = (struct AdjList\*) malloc(V \* sizeof(struct AdjList));**

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

**graph->array[i].head = NULL;**

**return graph;**

**}**

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

**struct AdjListNode\* newNode = newAdjListNode(dest);**

**newNode->next = graph->array[src].head;**

**graph->array[src].head = newNode;**

**newNode = newAdjListNode(src);**

**newNode->next = graph->array[dest].head;**

**graph->array[dest].head = newNode;**

**}**

**void printGraph(struct Graph\* graph) {**

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

**struct AdjListNode\* pCrawl = graph->array[v].head;**

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

**while (pCrawl) {**

**printf("-> %d", pCrawl->dest);**

**pCrawl = pCrawl->next;**

**}**

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

**}**

**}**

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

**int visited[MAX] = {0};**

**int queue[MAX];**

**int front = 0, rear = 0;**

**visited[startVertex] = 1;**

**queue[rear++] = startVertex;**

**printf("BFS traversal starting from vertex %d:\n", startVertex);**

**while (front < rear) {**

**int currentVertex = queue[front++];**

**printf("%d ", currentVertex);**

**struct AdjListNode\* temp = graph->array[currentVertex].head;**

**while (temp) {**

**int adjVertex = temp->dest;**

**if (!visited[adjVertex]) {**

**visited[adjVertex] = 1;**

**queue[rear++] = adjVertex;**

**}**

**temp = temp->next;**

**}**

**}**

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

**}**

**void DFSUtil(struct Graph\* graph, int vertex, int visited[]) {**

**visited[vertex] = 1;**

**printf("%d ", vertex);**

**struct AdjListNode\* temp = graph->array[vertex].head;**

**while (temp) {**

**int adjVertex = temp->dest;**

**if (!visited[adjVertex]) {**

**DFSUtil(graph, adjVertex, visited);**

**}**

**temp = temp->next;**

**}**

**}**

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

**int visited[MAX] = {0};**

**printf("DFS traversal starting from vertex %d:\n", startVertex);**

**DFSUtil(graph, startVertex, visited);**

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

**}**

**int main() {**

**int V = 5;**

**struct Graph\* graph = createGraph(V);**

**addEdge(graph, 0, 1);**

**addEdge(graph, 0, 4);**

**addEdge(graph, 1, 2);**

**addEdge(graph, 1, 3);**

**addEdge(graph, 1, 4);**

**addEdge(graph, 2, 3);**

**addEdge(graph, 3, 4);**

**printGraph(graph);**

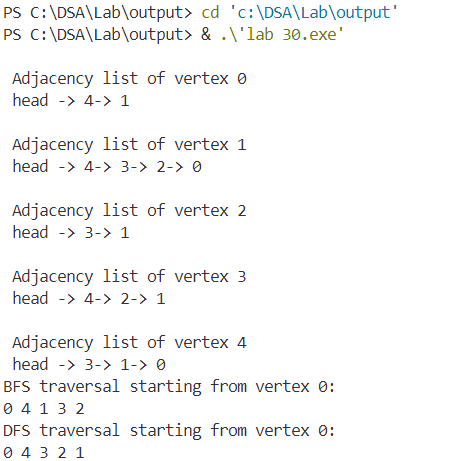
**BFS(graph, 0);**

**DFS(graph, 0);**

**return 0;**

**}**

**OUTPUT :**

****

**31. Write a program to implement Dijkstra’s Algorithm for finding single source shortest path problem.**

**INPUT :**

**#include <stdio.h>**

**#include <limits.h>**

**#include <stdbool.h>**

**#define V 9**

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

**int min = INT\_MAX, min\_index;**

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

**if (sptSet[v] == false && dist[v] <= min)**

**min = dist[v], min\_index = v;**

**return min\_index;**

**}**

**void printSolution(int dist[], int n) {**

**printf("Vertex\tDistance from Source\n");**

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

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

**}**

**void dijkstra(int graph[V][V], int src) {**

**int dist[V];**

**bool sptSet[V];**

**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, V);**

**}**

**int main() {**

**int graph[V][V] = {**

**{0, 4, 0, 0, 0, 0, 0, 8, 0},**

**{4, 0, 8, 0, 0, 0, 0, 11, 0},**

**{0, 8, 0, 7, 0, 4, 0, 0, 2},**

**{0, 0, 7, 0, 9, 14, 0, 0, 0},**

**{0, 0, 0, 9, 0, 10, 0, 0, 0},**

**{0, 0, 4, 14, 10, 0, 2, 0, 0},**

**{0, 0, 0, 0, 0, 2, 0, 1, 6},**

**{8, 11, 0, 0, 0, 0, 1, 0, 7},**

**{0, 0, 2, 0, 0, 0, 6, 7, 0}**

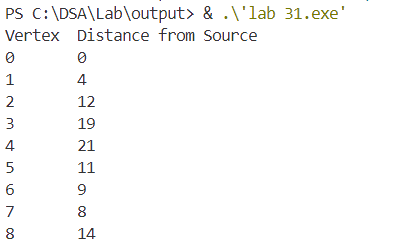
**};**

**dijkstra(graph, 0);**

**return 0;**

**}**

**OUTPUT :**

****

**32. Write a program to implement open addressing hashing approach.**

**INPUT :**

**#include <stdio.h>**

**#include <stdlib.h>**

**#define SIZE 10**

**void createHashTable(int hashTable[]) {**

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

**hashTable[i] = -1;**

**}**

**}**

**int hashFunction(int key) {**

**return key % SIZE;**

**}**

**void insert(int hashTable[], int key) {**

**int index = hashFunction(key);**

**while (hashTable[index] != -1) {**

**index = (index + 1) % SIZE;**

**}**

**hashTable[index] = key;**

**}**

**int search(int hashTable[], int key) {**

**int index = hashFunction(key);**

**int startIndex = index;**

**while (hashTable[index] != -1) {**

**if (hashTable[index] == key) {**

**return index;**

**}**

**index = (index + 1) % SIZE;**

**if (index == startIndex) {**

**return -1; // Element not found**

**}**

**}**

**return -1;**

**}**

**void displayHashTable(int hashTable[]) {**

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

**if (hashTable[i] != -1) {**

**printf("Index %d: %d\n", i, hashTable[i]);**

**} else {**

**printf("Index %d: Empty\n", i); } } }**

**int main() {**

**int hashTable[SIZE];**

**createHashTable(hashTable);**

**insert(hashTable, 10);**

**insert(hashTable, 20);**

**insert(hashTable, 30);**

**insert(hashTable, 25);**

**insert(hashTable, 35);**

**insert(hashTable, 50);**

**displayHashTable(hashTable);**

**int key = 25;**

**int result = search(hashTable, key);**

**if (result != -1) {**

**printf("Element %d found at index %d\n", key, result);**

**} else {**

**printf("Element %d not found in the hash table\n", key);**

**}**

**return 0;**

**}**

**OUTPUT :**

