Question 1: Write a program to simulate the working of stack using an array with the following: a) Push b) Pop c) Display The program should print appropriate messages for stack overflow, stack underflow

```c
#include <stdio.h>

#define MAX 5

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

void push(int item)
{
  if (top == MAX - 1)
  {
    printf("Stack Overflow! Cannot push %d\n", item);
    return;
  }
  stack[++top] = item;
  printf("%d pushed into stack\n", item);
}

void pop()
{
  if (top == -1)
  {
    printf("Stack Underflow! Stack is empty\n");
    return;
  }
  printf("%d popped from stack\n", stack[top--]);
}

void display()
{
  if (top == -1)
  {
    printf("Stack is empty\n");
    return;
```

```c
    printf("Stack elements (top to bottom): ");
    for (int i = top; i >= 0; i--)
      printf("%d ", stack[i]);
    printf("\n");
  }

  int main()
  {
    int choice, item;

    while (1)
    {
      printf("\n--- Stack Operations ---\n");
      printf("1. Push\n");
      printf("2. Pop\n");
      printf("3. Display\n");
      printf("4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch (choice)
      {
      case 1:
        printf("Enter element to push: ");
        scanf("%d", &item);
        push(item);
        break;

      case 2:
        pop();
        break;
```

```
        case 3:
            display();
            break;

        case 4:
            return 0;

        default:
            printf("Invalid choice!\n");
        }
    }
}
```

Output:

```
--- Stack Operations ---
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter element to push: 1
1 pushed into stack

--- Stack Operations ---
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter element to push: 3
3 pushed into stack

--- Stack Operations ---
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter element to push: 5
5 pushed into stack
```

```
--- Stack Operations ---
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2
5 popped from stack

--- Stack Operations ---
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack elements (top to bottom): 3 1

--- Stack Operations ---
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 4
```

Question 2(a): WAP to convert a given valid parenthesized infix arithmetic expression to postfix expression. The expression consists of single character operands and the binary operators + (plus), - (minus), * (multiply) and / (divide)

```
char stack[max];
int top = -1;

void push(char c)
{
   if (top == max - 1)
   {
      printf("Stack Overflow.");
      return;
   }
   stack[++top] = c;
}

char pop()
{
   if (top == -1)
   {
      printf("Stack Empty.");
      return '\0';
   }
   return stack[top--];
}

char peek()
{
   return stack[top];
}

int precedence(char op)
{
   if (op == '+' || op == '-')
      return 1;
   if (op == '*' || op == '/')
      return 2;
   return 0;
}
void infixToPostfix(char infix[])
{
  int k = 0;
  char c, postfix[max];

  for (int i = 0; infix[i] != '\0'; i++)
  {
    c = infix[i];
    if (isalnum(c))
    {
      postfix[k++] = c;
      postfix[k++] = ' ';
    }
    else if (c == '(')
    {
      push(c);
    }
    else if (c == ')')
    {
      while (top != -1 && peek() != '(')
      {
        postfix[k++] = pop();
        postfix[k++] = ' ';
      }
      pop();
    }
    else
    {
      while ((top != -1) && (precedence(peek()) >= precedence(c)))
      {
        postfix[k++] = pop();
        postfix[k++] = ' ';
```

```
            postfix[k++] = ' ';
          }
          push(c);
        }
    }
    while (top != -1)
    {
      postfix[k++] = pop();
      postfix[k++] = ' ';
    }
    postfix[k] = '\0';
    printf("Postfix Expression: %s", postfix);
}

int main()
{
    char infix[max];
    printf("Enter Expression: ");
    fgets(infix, max, stdin);
    infix[strcspn(infix, "\n")] = '\0';
    infixToPostfix(infix);
    return 0;
}
```

Output:

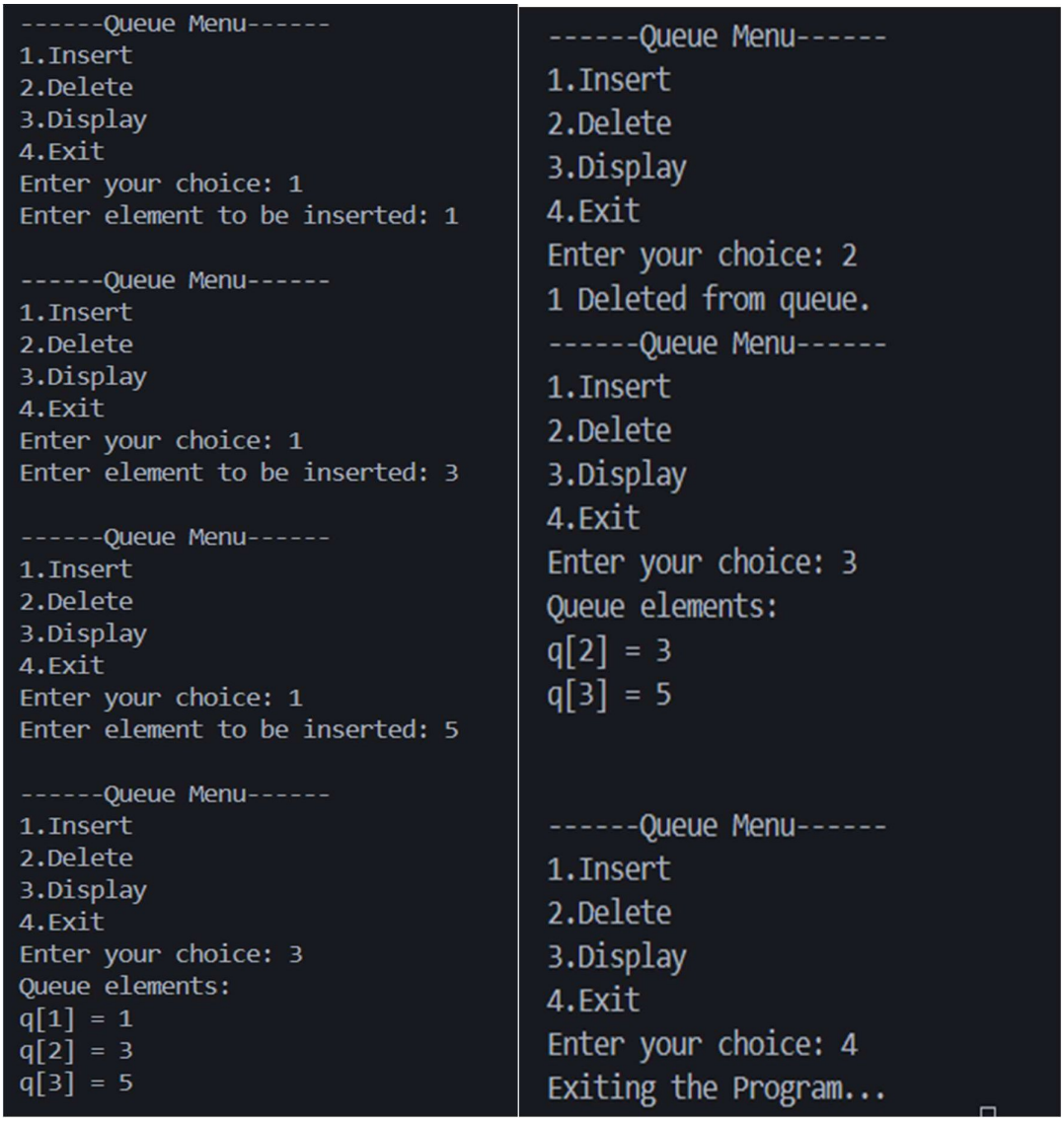
```
Enter Expression: (A+B)*(C-D)
Postfix Expression: A B + C D - *
```

Question 3(a): WAP to simulate the working of a queue of integers using an array. Provide the following operations: Insert, Delete, Display The program should print appropriate messages for queue empty and queue overflow conditions

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

int cqueue[size];
int front = -1, rear = -1;

void insert()
{
  int item;
  if (((front == 0) && (rear == size - 1)) || ((rear + 1) % size == front))
  {
    printf("Queue Overflow.");
    return;
  }
  printf("Enter element to be inserted: ");
  scanf("%d", &item);
  if (front == -1)
  {
    front = rear = 0;
  }
  else
  {
    rear = (rear + 1) % size;
  }
  cqueue[rear] = item;
}

void delete()
{
  if (front == -1)
  {
    printf("Queue Underflow.");
    return;
  }
  printf("%d Deleted from queue.", cqueue[front]);
  if (front == rear)
  {
    front = rear - 1;
  }
  else
  {
    front = (front + 1) % size;
  }
}

void display()
{
  if (front == -1)
  {
    printf("Queue empty.");
    return;
  }
  printf("Queue elements:\n");
  int i = front;
  while (1)
  {
    printf("q[%d] = %d\n", (i + 1), cqueue[i]);
    if (i == rear)
    {
      break;
    }
    i = (i + 1) % size;
  }
  printf("\n");
}
```

```c
int main()
{
  int choice;
  while (1)
  {
    printf("\n------Queue Menu------\n");
    printf("1.Insert\n2.Delete\n3.Display\n4.Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice)
    {
    case 1:
      insert();
      break;
    case 2:
      delete();
      break;
    case 3:
      display();
      break;
    case 4:
      return 0;
    default:
      printf("Invalid input.\n");
    }
  }
}
```

Output:


------Queue Menu------
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice: 1
Enter element to be inserted: 1
------Queue Menu------
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice: 1
Enter element to be inserted: 3
------Queue Menu------
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice: 1
Enter element to be inserted: 5
------Queue Menu------
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice: 3
Queue elements:
q[1] = 1
q[2] = 3
q[3] = 5
------Queue Menu------
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice: 2
1 Deleted from queue.
------Queue Menu------
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice: 3
Queue elements:
q[2] = 3
q[3] = 5
------Queue Menu------
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice: 4
Exiting the Program...

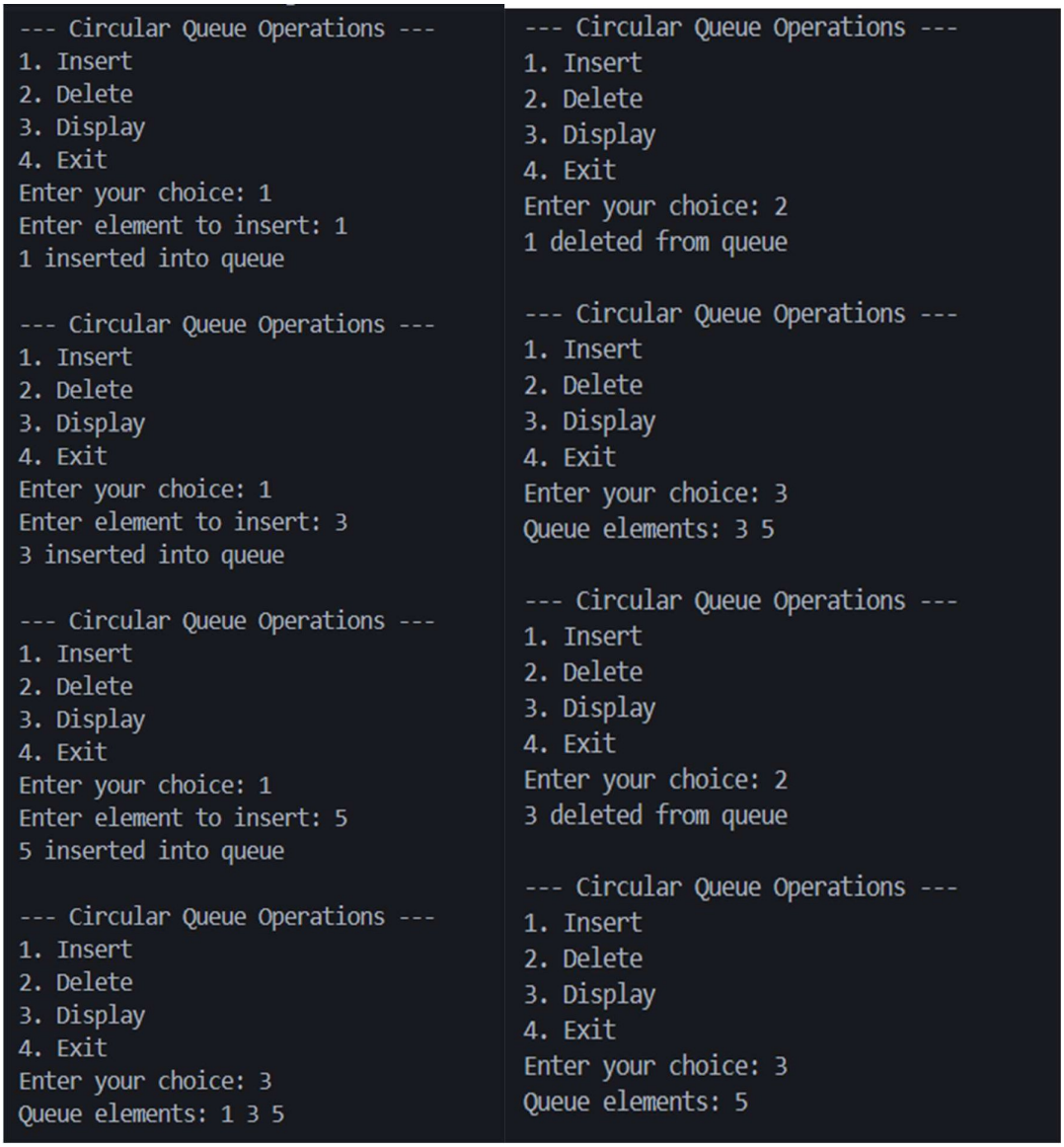
Question 3(b): WAP to simulate the working of a circular queue of integers using an array. Provide the following operations: Insert, Delete & Display The program should print appropriate messages for queue empty and queue overflow conditions

```c
#define MAX 5

int cq[MAX];
int front = -1, rear = -1;

void insert(int item)
{
  if ((rear + 1) % MAX == front)
  {
    printf("Queue Overflow! Circular Queue is full\n");
    return;
  }

  if (front == -1)
  {
    front = rear = 0;
  }
  else
  {
    rear = (rear + 1) % MAX;
  }

  cq[rear] = item;
  printf("%d inserted into queue\n", item);
}

void delete()
{
  if (front == -1)
  {
    printf("Queue Empty! Nothing to delete\n");
    return;
  }

  printf("%d deleted from queue\n", cq[front]);
```

```c
void delete()
{
  if (front == rear)
  {
    front = rear = -1;
  }
  else
  {
    front = (front + 1) % MAX;
  }
}

void display()
{
  int i;

  if (front == -1)
  {
    printf("Queue is empty\n");
    return;
  }

  printf("Queue elements: ");
  i = front;
  while (1)
  {
    printf("%d ", cq[i]);
    if (i == rear)
      break;
    i = (i + 1) % MAX;
  }
  printf("\n");
}
```

```c
int main()
{
  while (1)
  {
    printf("\n--- Circular Queue Operations ---\n");
    printf("1. Insert\n");
    printf("2. Delete\n");
    printf("3. Display\n");
    printf("4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice)
    {
    case 1:
      printf("Enter element to insert: ");
      scanf("%d", &item);
      insert(item);
      break;

    case 2:
      delete();
      break;

    case 3:
      display();
      break;

    case 4:
      return 0;

    default:
      printf("Invalid choice!\n");
    }
  }
}
```

Output:



```
--- Circular Queue Operations ---
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter element to insert: 1
1 inserted into queue

--- Circular Queue Operations ---
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter element to insert: 3
3 inserted into queue

--- Circular Queue Operations ---
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter element to insert: 5
5 inserted into queue

--- Circular Queue Operations ---
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
Queue elements: 1 3 5
```

```
--- Circular Queue Operations ---
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2
1 deleted from queue

--- Circular Queue Operations ---
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
Queue elements: 3 5

--- Circular Queue Operations ---
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2
3 deleted from queue

--- Circular Queue Operations ---
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
Queue elements: 5
```

```
--- Circular Queue Operations ---
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter element to insert: 7
7 inserted into queue

--- Circular Queue Operations ---
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
Queue elements: 5 7

--- Circular Queue Operations ---
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 4
```

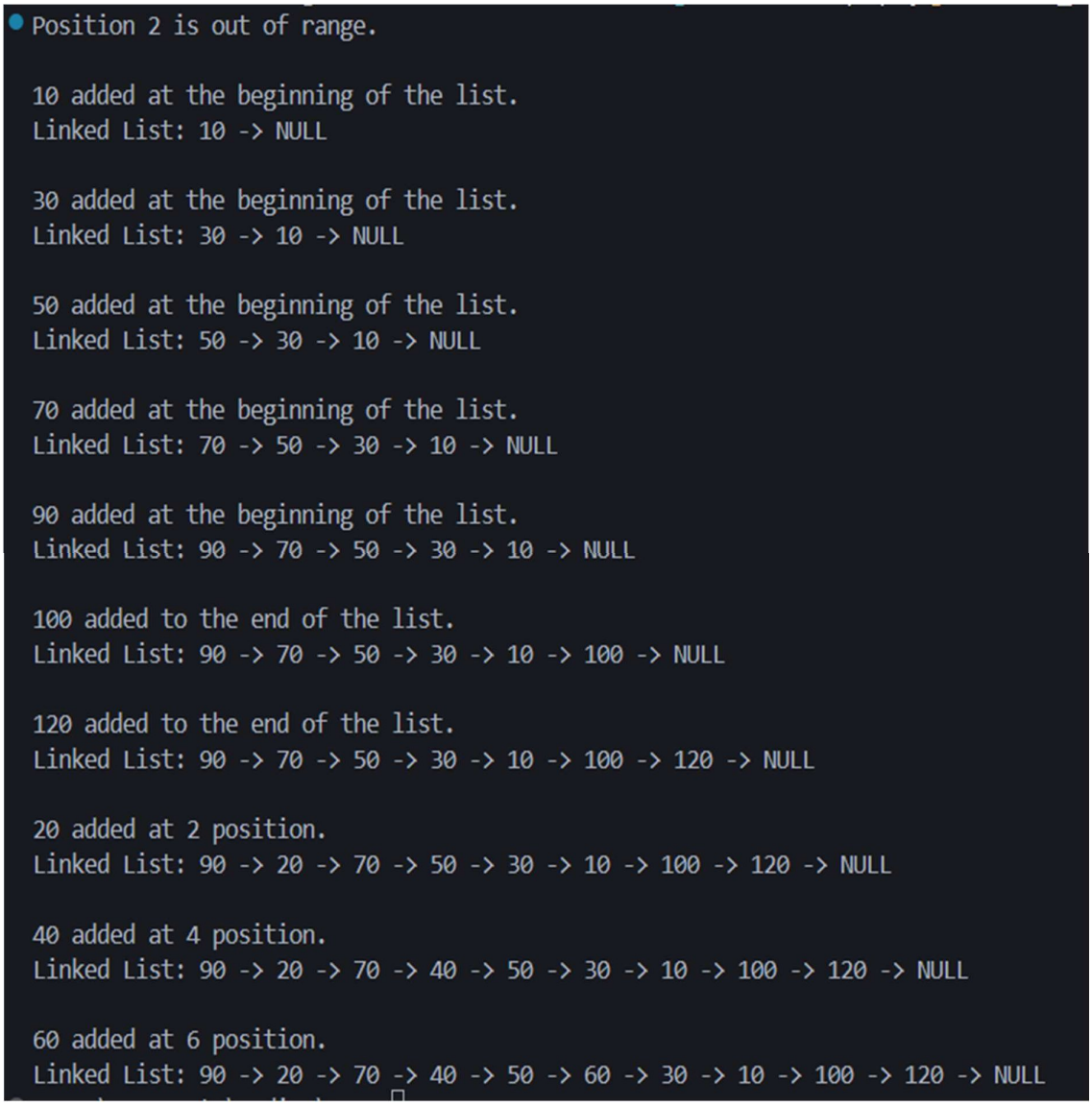
Question 4(a): WAP to Implement Singly Linked List with following operations a) Create a linked list. b) Insertion of a node at first position, at any position and at end of list. Display the contents of the linked list.

```c
struct Node
{
  int data;
  struct Node *next;
};

struct Node *createNode(int value)
{
  struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
  newNode->data = value;
  newNode->next = NULL;
  return newNode;
}

void display(struct Node *head)
{
  struct Node *temp = head;
  printf("Linked List: ");
  while (temp != NULL)
  {
    printf("%d -> ", temp->data);
    temp = temp->next;
  }
  printf("NULL\n");
}

void insertAtBeg(struct Node **head, int value)
{
  struct Node *newNode = createNode(value);
  newNode->next = *head;
  *head = newNode;
  printf("\n%d added at the beginning of the list.\n", value);
  display(*head);
}
```

```c
void insertAtEnd(struct Node **head, int value)
{
  struct Node *newNode = createNode(value);
  if (*head == NULL)
  {
    *head = newNode;
    printf("\n%d added to the end of the list.\n", value);
    display(*head);
    return;
  }
  struct Node *temp = *head;
  while (temp->next != NULL)
  {
    temp = temp->next;
  }
  temp->next = newNode;
  printf("\n%d added to the end of the list.\n", value);
  display(*head);
}

void insertAtPos(struct Node **head, int value, int pos)
{
  struct Node *newNode = createNode(value);
  if (pos == 1)
  {
    free(newNode);
    insertAtBeg(head, value);
    return;
  }
  struct Node *temp = *head;
  for (int i = 1; i < pos - 1 && temp != NULL; i++)
  {
    temp = temp->next;
  }
  if (temp == NULL)
```

```c
    printf("Position %d is out of range.\n", pos);
    free(newNode);
    return;
  }
  newNode->next = temp->next;
  temp->next = newNode;
  printf("\n%d added at %d position.\n", value, pos);
  display(*head);
}

int main()
{
  struct Node *head = NULL;
  insertAtPos(&head, 20, 2);

  insertAtBeg(&head, 10);
  insertAtBeg(&head, 30);
  insertAtBeg(&head, 50);
  insertAtBeg(&head, 70);
  insertAtBeg(&head, 90);

  insertAtEnd(&head, 100);
  insertAtEnd(&head, 120);

  insertAtPos(&head, 20, 2);
  insertAtPos(&head, 40, 4);
  insertAtPos(&head, 60, 6);

  return 0;
}
```

Output:


Position 2 is out of range.
10 added at the beginning of the list.
Linked List: 10 -> NULL
30 added at the beginning of the list.
Linked List: 30 -> 10 -> NULL
50 added at the beginning of the list.
Linked List: 50 -> 30 -> 10 -> NULL
70 added at the beginning of the list.
Linked List: 70 -> 50 -> 30 -> 10 -> NULL
90 added at the beginning of the list.
Linked List: 90 -> 70 -> 50 -> 30 -> 10 -> NULL
100 added to the end of the list.
Linked List: 90 -> 70 -> 50 -> 30 -> 10 -> 100 -> NULL
120 added to the end of the list.
Linked List: 90 -> 70 -> 50 -> 30 -> 10 -> 100 -> 120 -> NULL
20 added at 2 position.
Linked List: 90 -> 20 -> 70 -> 50 -> 30 -> 10 -> 100 -> 120 -> NULL
40 added at 4 position.
Linked List: 90 -> 20 -> 70 -> 40 -> 50 -> 30 -> 10 -> 100 -> 120 -> NULL
60 added at 6 position.
Linked List: 90 -> 20 -> 70 -> 40 -> 50 -> 60 -> 30 -> 10 -> 100 -> 120 -> NULL

Question 4(b): Given two strings s and part, perform the following operation on s until **all** occurrences of the substring part are removed:

- Find the **leftmost** occurrence of the substring part and **remove** it from s. Return s *after removing all occurrences of* part.
  A **substring** is a contiguous sequence of characters in a string.

```c
char* removeOccurrences(char* s, char* part) {
    int n = strlen(s);
    int m = strlen(part);

    int k = 0;

    for (int i = 0; i < n; i++) {
        s[k++] = s[i];

        if (k >= m && strncmp(s + k - m, part, m) == 0) {
            k -= m;
        }
    }

    s[k] = '\0';
    return s;
}
```

Saved

Testcase | Test Result

part =

"abc"

Output

"dab"

Expected

"dab"

Question 5(a): WAP to Implement Singly Linked List with following operations a) Create a linked list. b) Deletion of first element, specified element and last element in the list. c) Display the contents of the linked list.

```c
struct NOde
{
  int data;
  struct NOde *next;
};

struct NOde *head = NULL;

void insert(int value)
{
  struct NOde *newNOde = (struct NOde *)malloc(sizeof(struct NOde));
  newNOde->data = value;
  newNOde->next = NULL;

  if (head == NULL)
  {
    head = newNOde;
    return;
  }

  struct NOde *temp = head;
  while (temp->next != NULL)
  {
    temp = temp->next;
  }
  temp->next = newNOde;
}

void display()
{
  struct NOde *temp = head;
  printf("Linked List: ");
  while (temp != NULL)
  {
    printf("%d -> ", temp->data);
```

```c
void deleteFirst()
{
  if (head == NULL)
  {
    printf("List is empty.");
    return;
  }
  struct NOde *temp = head;
  head = head->next;
  printf("\nDeleted 1st element: %d.\n", temp->data);
}

void deleteLast()
{
  if (head == NULL)
  {
    printf("List is empty.");
    return;
  }
  struct NOde *temp = head;
  struct NOde *prev = NULL;

  while (temp->next != NULL)
  {
    prev = temp;
    temp = temp->next;
  }

  if (prev == NULL)
  {
    head = NULL;
  }

  else
  {
```

```c
void deleteLast()
{
  printf("\nDeleted last element: %d.\n", temp->data);
  display();
  free(temp);
}

void deleteSp(int value)
{
  if (head == NULL)
  {
    printf("list is empty.");
    return;
  }
  struct NOde *temp = head;
  struct NOde *prev = NULL;

  while (temp != NULL && temp->data != value)
  {
    prev = temp;
    temp = temp->next;
  }
  if (temp == NULL)
  {
    printf("Element %d not found.", value);
    return;
  }

  if (prev == NULL)
  {
    head = head->next;
  }
  else
  {
    prev->next = temp->next;
  }
```

```c
void deleteSp(int value)
  else
  {
    printf("\nDeleted element: %d\n", temp->data);
    display();
    free(temp);
}

int main()
{
  insert(10);
  insert(30);
  insert(50);
  insert(70);
  insert(90);
  insert(110);
  insert(130);

  deleteFirst();
  deleteFirst();

  deleteLast();

  deleteSp(90);

  return 0;
}
```

Output:

```
Deleted 1st element: 10.

Deleted 1st element: 30.

Deleted last element: 130.
Linked List: 50 -> 70 -> 90 -> 110 -> NULL

Deleted element: 90
Linked List: 50 -> 70 -> 110 -> NULL
```

Question 5(b): On day 1, one person discovers a secret. You are given an integer delay, which means that each person will **share** the secret with a new person **every day**, starting from delay days after discovering the secret. You are also given an integer forget, which means that each person will **forget** the secret forget days after discovering it. A person **cannot** share the secret on the same day they forgot it, or on any day afterwards.

Given an integer n, return *the number of people who know the secret at the end of day* n. Since the answer may be very large, return it **modulo** $10^9 + 7$.

```
int peopleAwareOfSecret(int n, int delay, int forget){
    const long long MOD = 1000000007LL;
    long long *dp = (long long*)calloc(n+1, sizeof(long long));
    long long *pref = (long long*)calloc(n+1, sizeof(long long));
    dp[1] = 1;
    pref[1] = 1;
    for(int i = 2; i <= n; ++i){
        int left = i - forget;
        if(left < 0) left = 0;
        int right = i - delay;
        if(right >= 1){
            long long sum = (pref[right] - pref[left] + MOD) % MOD;
            dp[i] = sum;
        } else {
            dp[i] = 0;
        }
        pref[i] = (pref[i-1] + dp[i]) % MOD;
    }
    int cutoff = n - forget;
    if(cutoff < 0) cutoff = 0;
    long long ans = (pref[n] - pref[cutoff] + MOD) % MOD;
    free(dp);
    free(pref);
    return (int)ans;
}
```

Accepted Runtime: 0 ms

Case 1 Case 2

Input

n =
6

delay =
2

forget =
4

Output

5

Expected

5

Question 6(a): WAP to Implement Single Link List with following operations: Sort the linked list, Reverse the linked list, Concatenation of two linked lists.

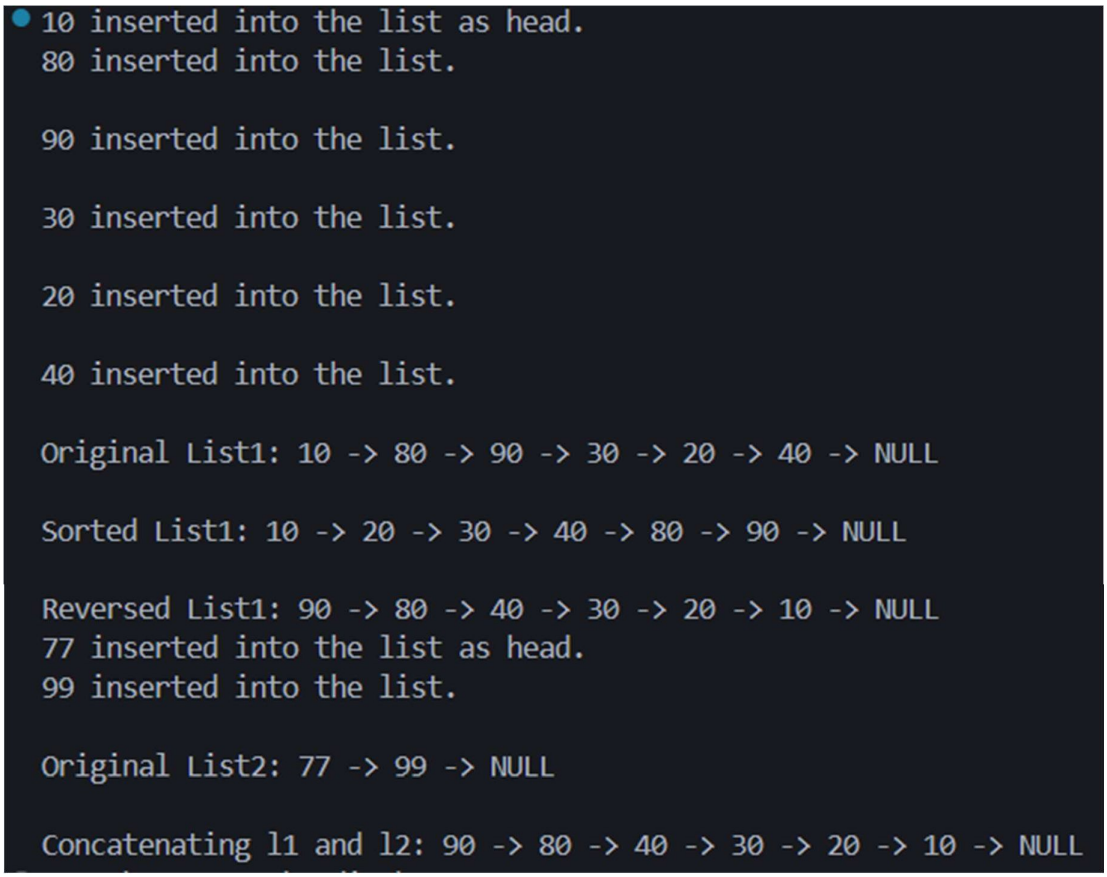
```c
struct NOde
{
  int data;
  struct NOde *next;
};

typedef struct NOde NOde;

NOde *createNOde(int value)
{
  NOde *newNOde = (struct NOde *)malloc(sizeof(struct NOde));
  if (newNOde == NULL)
  {
    printf("Error allocating memory.");
    return newNOde;
  }

  newNOde->data = value;
  newNOde->next = NULL;
  return newNOde;
}

void display(NOde *head)
{
  NOde *temp = head;
  if (temp == NULL)
  {
    printf("List is empty.");
    return;
  }
  while (temp != NULL)
  {
    printf("%d -> ", temp->data);
    temp = temp->next;
  }
```

```c
void insert(struct NOde **head, int value)
  NOde *newNOde = createNOde(value);
  if (*head == NULL)
  {
    *head = newNOde;
    printf("%d inserted into the list as head.", value);
    return;
  }
  NOde *temp = *head;
  while (temp->next != NULL)
  {
    temp = temp->next;
  }
  temp->next = newNOde;
  printf("\n%d inserted into the list.\n", value);
}

void sort(NOde *head)
{
  if (head == NULL)
  {
    printf("List is empty.");
    return;
  }

  int swapped;
  NOde *ptr, *lptr = NULL;
  do
  {
    swapped = 0;
    ptr = head;
    while (ptr->next != lptr)
    {
      if (ptr->data > ptr->next->data)
      {
```

```
        if (ptr->data > ptr->next->data)
        {
          int temp = ptr->data;
          ptr->data = ptr->next->data;
          ptr->next->data = temp;

          swapped = 1;
        }
        ptr = ptr->next;
      }
      lptr = ptr;
    } while (swapped);
}

void reverse(NOde **head)
{
  struct NOde *prev = NULL, *curr = *head, *next = NULL;
  while (curr != NULL)
  {
    next = curr->next;
    curr->next = prev;
    prev = curr;
    curr = next;
  }
  *head = prev;
}

void concat(NOde **head1, NOde **head2)
{
  if (*head1 == NULL)
  {
    *head1 = *head2;
    return;
```

```
void concat(NOde **head1, NOde **head2)
  NOde *temp = *head2;
  while (temp->next != NULL)
    temp = temp->next;
  temp->next = *head2;
}

int main()
{
  NOde *l1 = NULL;
  NOde *l2 = NULL;
  insert(&l1, 10);
  insert(&l1, 80);
  insert(&l1, 90);
  insert(&l1, 30);
  insert(&l1, 20);
  insert(&l1, 40);
  printf("\nOriginal List1: ");
  display(l1);
  sort(l1);
  printf("\nSorted List1: ");
  display(l1);
  reverse(&l1);
  printf("\nReversed List1: ");
  display(l1);

  insert(&l2, 77);
  insert(&l2, 99);
  printf("\nOriginal List2: ");
  display(l2);
  printf("\nConcatenating l1 and l2: ");
  concat(&l1, &l2);
  display(l1);
  return 0;
}
```

Output:


10 inserted into the list as head.
80 inserted into the list.
90 inserted into the list.
30 inserted into the list.
20 inserted into the list.
40 inserted into the list.
Original List1: 10 -> 80 -> 90 -> 30 -> 20 -> 40 -> NULL
Sorted List1: 10 -> 20 -> 30 -> 40 -> 80 -> 90 -> NULL
Reversed List1: 90 -> 80 -> 40 -> 30 -> 20 -> 10 -> NULL
77 inserted into the list as head.
99 inserted into the list.
Original List2: 77 -> 99 -> NULL
Concatenating l1 and l2: 90 -> 80 -> 40 -> 30 -> 20 -> 10 -> NULL

Question 5(b): WAP to Implement Single Link List to simulate Stack & Queue Operations.

```c
struct Node
{
  int data;
  struct Node *next;
};

typedef struct Node Node;

Node *createNode(int data)
{
  Node *newNode = (Node *)malloc(sizeof(Node));
  if (newNode == NULL)
  {
    printf("Error: Memory Allocation failed.");
    return NULL;
  }
  newNode->data = data;
  newNode->next = NULL;
  return newNode;
}

void display(Node *head)
{
  Node *temp = head;
  if (temp == NULL)
  {
    printf("List empty.\n");
    return;
  }
  while (temp != NULL)
  {
    printf("%d -> ", temp->data);
    temp = temp->next;
  }
  printf("NULL\n");
```

```c
void push(Node **top, int data)
  Node *newNode = createNode(data);
  newNode->next = *top;
  *top = newNode;
  printf("%d pushed into stack.\n", data);
}

int pop(Node **top)
{
  if (*top == NULL)
  {
    printf("Stack empty.");
    return -1;
  }
  Node *temp = *top;
  int poppedData = temp->data;
  *top = (*top)->next;
  free(temp);
  return poppedData;
}

// Queue Operations

struct Queue
{
  Node *front;
  Node *rear;
};

typedef struct Queue Queue;

Queue *createQueue()
{
  Queue *q = (Queue *)malloc(sizeof(Queue));
  if (q == NULL)
```

```
Queue *createQueue()
  if (q == NULL)
    printf("Error: Memory Allocation failed.\n");
    return NULL;
  }
  q->front = q->rear = NULL;
  return q;
}

void enQ(Queue *q, int data)
{
  Node *newNode = createNode(data);
  if (q->rear == NULL)
  {
    q->front = q->rear = newNode;
  }
  else
  {
    q->rear->next = newNode;
    q->rear = newNode;
  }
  printf("%d enqueued into queue.\n", data);
}

int deQ(Queue *q)
{
  if (q->front == NULL)
  {
    printf("Queue is empty.");
    return -1;
  }
  Node *temp = q->front;
  int deqData = temp->data;
  q->front = q->front->next;
  if (q->front == NULL)
```

```
int main()
{
  printf("Stack Operations:\n");
  Node *stackTop = NULL;
  push(&stackTop, 10);
  push(&stackTop, 20);
  push(&stackTop, 30);
  printf("Stack: ");
  display(stackTop);
  printf("Poppped %d\n", pop(&stackTop));
  printf("Stack: ");
  display(stackTop);
  printf("Poppped %d\n", pop(&stackTop));
  printf("Stack: ");
  display(stackTop);

  printf("Queue operations:\n");
  Queue *q = createQueue();
  enQ(q, 100);
  enQ(q, 200);
  enQ(q, 300);
  printf("Queue: ");
  display(q->front);
  printf("Dequeued %d\n", deQ(q));
  printf("Dequeued %d\n", deQ(q));
  printf("Queue: ");
  display(q->front);
  enQ(q, 400);
  printf("Queue: ");
  display(q->front);
  free(q);

  return 0;
```

Output:

```
Stack Operations:
10 pushed into stack.
20 pushed into stack.
30 pushed into stack.
Stack: 30 -> 20 -> 10 -> NULL
Poppped 30
Stack: 20 -> 10 -> NULL
Poppped 20
Stack: 10 -> NULL
Queue operations:
100 enqueued into queue.
200 enqueued into queue.
300 enqueued into queue.
Queue: 100 -> 200 -> 300 -> NULL
Dequeued 100
Dequeued 200
Queue: 300 -> NULL
400 enqueued into queue.
Queue: 300 -> 400 -> NULL
```

Question 7(a): WAP to Implement doubly link list with primitive operations a) Create a doubly linked list. b) Insert a new node to the left of the node. c) Delete the node based on a specific value d) Display the contents of the list

```c
struct NOde
{
  int data;
  struct NOde *next, *prev;
};

struct NOde *createNode(int value)
{
  struct NOde *newNOde = (struct NOde *)malloc(sizeof(struct NOde));
  newNOde->data = value;
  newNOde->next = NULL;
  newNOde->prev = NULL;
  return newNOde;
}

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

// insertion

void insertAtEnd(struct NOde **head, int value)
{
  struct NOde *newNOde = createNode(value);
  if (*head == NULL)
  {
    *head = newNOde;
    return;
  }
  struct NOde *temp = *head;
  while (temp->next != NULL)
  {
    temp = temp->next;
  }
  newNOde->next = NULL;
  newNOde->prev = temp;
  temp->next = newNOde;
  printf("\n%d inserted at the end of the list.\n", value);
}

// deletion

void deleteValue(struct NOde **head, int value)
{
  struct NOde *ptr, *temp = *head;
  int val;
  while (temp->data != value && temp != NULL)
  {
    temp = temp->next;
  }
  if (temp == NULL)
  {
    printf("value not found.\n");
    return;
  }
  if (temp->prev == NULL)
  {
    *head = temp->next;
    if (*head)
      (*head)->prev = NULL;
  }
```

```
  }
  else
  {
    temp->prev->next = temp->next;
  }
  if (temp->next)
    temp->next->prev = temp->prev;

  printf("\n%d deleted from the list.\n", value);
  free(temp);
}

int main()
{
  struct NOde *head = NULL;
  insertAtPos(&head, 20, 2);

  insertAtBeg(&head, 10);
  insertAtBeg(&head, 30);
  insertAtBeg(&head, 20);
  insertAtBeg(&head, 40);
  insertAtBeg(&head, 50);
  insertAtBeg(&head, 70);
  insertAtBeg(&head, 60);
  insertAtBeg(&head, 90);

  insertAtEnd(&head, 10);
  insertAtEnd(&head, 20);
  insertAtEnd(&head, 30);

  insertAtPos(&head, 80, 2);
  insertAtPos(&head, 100, 10);
  insertAtPos(&head, 110, 11);
```

```
  insertAtEnd(&head, 10);
  insertAtEnd(&head, 20);
  insertAtEnd(&head, 30);

  insertAtPos(&head, 80, 2);
  insertAtPos(&head, 100, 10);
  insertAtPos(&head, 110, 11);

  display(head);
  deleteAtBeg(&head);
  display(head);
  deleteAtEnd(&head);
  display(head);
  deleteValue(&head, 40);
  display(head);

  sort(head);
  display(head);

  return 0;
}
```

Output:

```
30 inserted at the end of the list.

20 inserted at the end of the list.

40 inserted at the end of the list.

50 inserted at the end of the list.

70 inserted at the end of the list.

60 inserted at the end of the list.

90 inserted at the end of the list.

10 inserted at the end of the list.

20 inserted at the end of the list.

30 inserted at the end of the list.
10 -> 30 -> 20 -> 40 -> 50 -> 70 -> 60 -> 90 -> 10 -> 20 -> 30 -> NULL

40 deleted from the list.
10 -> 30 -> 20 -> 50 -> 70 -> 60 -> 90 -> 10 -> 20 -> 30 -> NULL
```

Question 7(b): Given a singly linked list, return a random node's value from the linked list. Each node must have the **same probability** of being chosen.
Implement the Solution class:

- Solution(ListNode head) Initializes the object with the head of the singly-linked list head.
- int getRandom() Chooses a node randomly from the list and returns its value. All the nodes of the list should be equally likely to be chosen.

C Auto

```c
#include <stdlib.h>

typedef struct {
    struct ListNode *head;
} Solution;

Solution* solutionCreate(struct ListNode* head) {
    Solution* obj = malloc(sizeof(Solution));
    obj->head = head;
    return obj;
}

int solutionGetRandom(Solution* obj) {
    struct ListNode *cur = obj->head;
    int res = cur->val;
    int i = 1;
    while (cur) {
        if (rand() % i == 0) res = cur->val;
        cur = cur->next;
        i++;
    }
    return res;
}

void solutionFree(Solution* obj) {
    free(obj);
}
```

Case 1

Input

```
["Solution","getRandom","getRandom","getRandom","getRandom","getRandom"]
```

```
[[[1,2,3]],[],[],[],[],[]]
```

Output

```
[null,3,1,3,2,1]
```

Expected

```
[null,2,2,1,2,3]
```

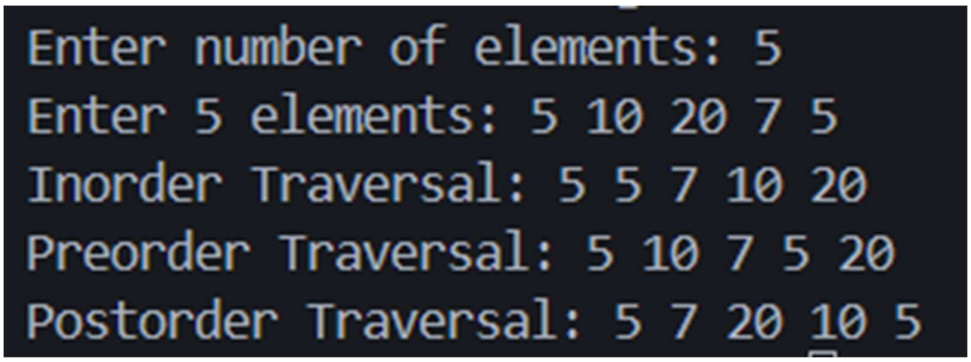
Question 8(a): Write a program a) To construct a binary Search tree. b) To traverse the tree using all the methods i.e., in-order, preorder and post order c) To display the elements in the tree.

```c
typedef struct Node
{
  int data;
  struct Node *left, *right;
} n;

n *newnode(int val)
{
  n *node = malloc(sizeof *node);
  if (node == NULL)
  {
    perror("malloc");
    exit(EXIT_FAILURE);
  }
  node->data = val;
  node->left = node->right = NULL;
  return node;
}

n *insert(n *root, int val)
{
  if (root == NULL)
    return newnode(val);
  if (val < root->data)
    root->left = insert(root->left, val);
  else
    root->right = insert(root->right, val);
  return root;
}

void inorder(n *root)
{
  if (root == NULL)
    return;
  inorder(root->left);
```

```c
void inorder(n *root)
  printf("%d ", root->data);
  inorder(root->right);
}

void preorder(n *root)
{
  if (root == NULL)
    return;
  printf("%d ", root->data);
  preorder(root->left);
  preorder(root->right);
}

void postorder(n *root)
{
  if (root == NULL)
    return;
  postorder(root->left);
  postorder(root->right);
  printf("%d ", root->data);
}

int main()
{
  n *root = NULL;
  int count, val;
  printf("Enter number of elements: ");
  if (scanf("%d", &count) != 1)
    return 1;

  printf("Enter %d elements: ", count);
  for (int i = 0; i < count; i++)
  {
    scanf("%d", &val);
```

```c
int main()
{
  n *root = NULL;
  int count, val;
  printf("Enter number of elements: ");
  if (scanf("%d", &count) != 1)
    return 1;

  printf("Enter %d elements: ", count);
  for (int i = 0; i < count; i++)
  {
    scanf("%d", &val);
    root = insert(root, val);
  }
  printf("Inorder Traversal: ");
  inorder(root);
  printf("\n");
  printf("Preorder Traversal: ");
  preorder(root);
  printf("\n");
  printf("Postorder Traversal: ");
  postorder(root);
  printf("\n");
  return 0;
}
```

Output:


Enter number of elements: 5
Enter 5 elements: 5 10 20 7 5
Inorder Traversal: 5 5 7 10 20
Preorder Traversal: 5 10 7 5 20
Postorder Traversal: 5 7 20 10 5

Question 8(b): You are given the head of a linked list containing unique integer values and an integer array nums that is a subset of the linked list values. Return *the number of connected components in* nums *where two values are connected if they appear consecutively in the linked list*.

```c
int cmp_int(const void *a, const void *b) {
    int x = *(const int*)a;
    int y = *(const int*)b;
    if (x < y) return -1;
    if (x > y) return 1;
    return 0;
}

int bs(int *arr, int n, int target) {
    int l = 0, r = n - 1;
    while (l <= r) {
        int m = l + (r - l) / 2;
        if (arr[m] == target) return 1;
        if (arr[m] < target) l = m + 1;
        else r = m - 1;
    }
    return 0;
}

int numComponents(struct ListNode* head, int* nums, int numsSize) {
    if (numsSize == 0) return 0;
    qsort(nums, numsSize, sizeof(int), cmp_int);
    int count = 0;
    int prev_in = 0;
    struct ListNode *cur = head;
    while (cur) {
        int in = bs(nums, numsSize, cur->val);
        if (in && !prev_in) count++;
```

Testcase | Test Result

Accepted Runtime: 0 ms

Case 1 Case 2

Input

head =

[0,1,2,3]

nums =

[0,1,3]

Output

2

Expected

2

Question 9(a): Write a program to traverse a graph using BFS method.

```c
#define MAX 20

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

void enqueue(int v)
{
  if (rear == MAX - 1)
    return;
  if (front == -1)
    front = 0;
  queue[++rear] = v;
}

int dequeue()
{
  if (front == -1 || front > rear)
    return -1;
  return queue[front++];
}

void BFS(int a[MAX][MAX], int n, int start)
{
  int i, v;
  for (i = 0; i < n; i++)
    visited[i] = 0;

  enqueue(start);
  visited[start] = 1;

  printf("BFS Traversal: ");

  while ((v = dequeue()) != -1)
  {
    printf("%d ", v);
    for (i = 0; i < n; i++)
    {
      if (a[v][i] == 1 && !visited[i])
      {
        enqueue(i);
        visited[i] = 1;
      }
    }
  }
}

int main()
{
  int n, i, j, start;
  int a[MAX][MAX];

  printf("Enter number of vertices: ");
  scanf("%d", &n);

  printf("Enter adjacency matrix:\n");
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      scanf("%d", &a[i][j]);

  printf("Enter starting vertex: ");
  scanf("%d", &start);

  BFS(a, n, start);

  return 0;
}
```

Output:

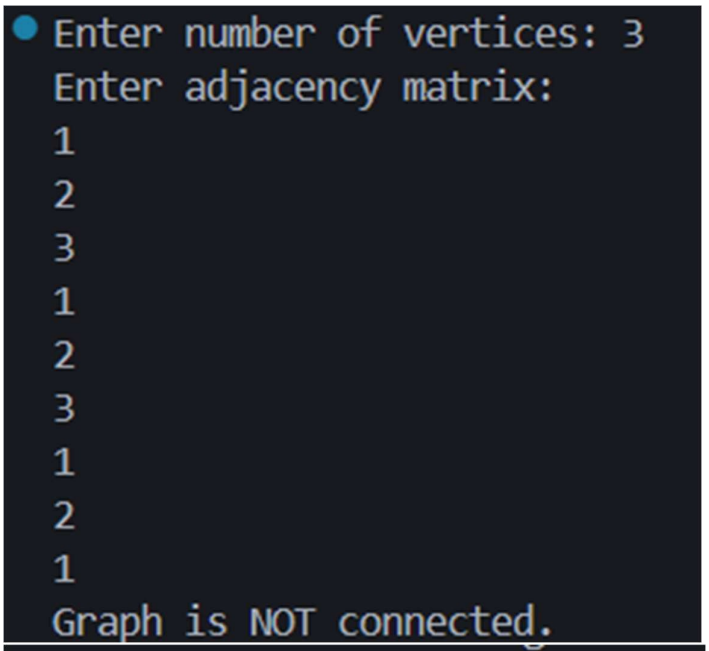
```
Enter number of vertices: 3
Enter adjacency matrix:
1
2
1
3
1
2
3
1
2
Enter starting vertex: 2
BFS Traversal: 2 1
```

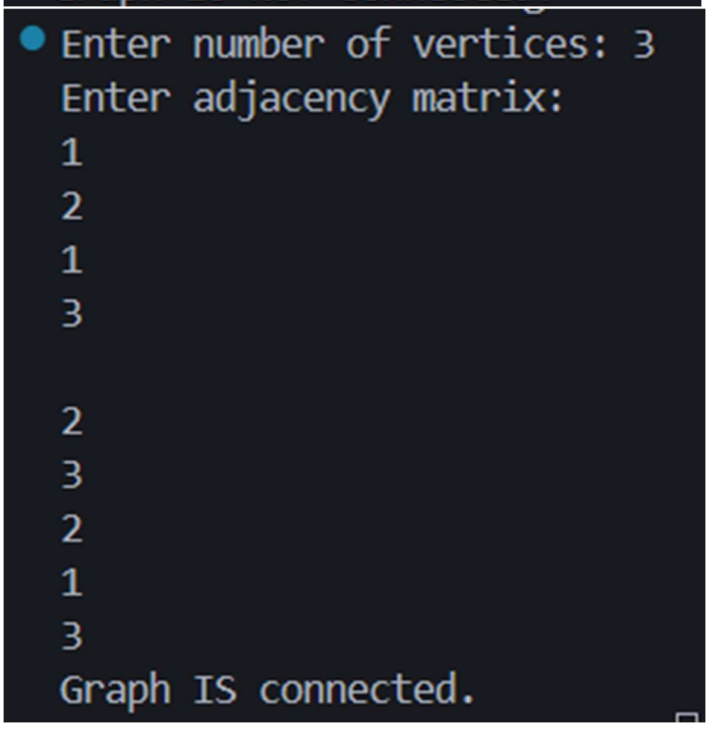
Question 9(b): Write a program to check whether given graph is connected or not using DFS method.

```c
#define MAX 20

int visited[MAX];

void DFS(int a[MAX][MAX], int n, int v)
{
  visited[v] = 1;
  for (int i = 0; i < n; i++)
  {
    if (a[v][i] == 1 && !visited[i])
      DFS(a, n, i);
  }
}

int main()
{
  int n, i, j;
  int a[MAX][MAX];

  printf("Enter number of vertices: ");
  scanf("%d", &n);

  printf("Enter adjacency matrix:\n");
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      scanf("%d", &a[i][j]);

  for (i = 0; i < n; i++)
    visited[i] = 0;

  DFS(a, n, 0); // Start from vertex 0

  for (i = 0; i < n; i++)
  {
    if (!visited[i])
```

```c
  for (i = 0; i < n; i++)
  {
    if (!visited[i])
    {
      printf("Graph is NOT connected.\n");
      return 0;
    }
  }

  printf("Graph IS connected.\n");
  return 0;
}
```

Output:


Enter number of vertices: 3
Enter adjacency matrix:
1
2
3
1
2
3
1
2
1
Graph is NOT connected.


Enter number of vertices: 3
Enter adjacency matrix:
1
2
1
3
2
3
2
1
3
Graph IS connected.

Question 10: Given a File of N employee records with a set K of Keys(4digit) which uniquely determine the records in file F. Assume that file F is maintained in memory by a Hash Table (HT) of m memory locations with L as the set of memory addresses (2-digit) of locations in HT. Let the keys in K and addresses in L are integers. Design and develop a Program in C that uses Hash function H: K -> L as H(K)=K mod m (remainder method), and implement hashing technique to map a given key K to the address space L. Resolve the collision (if any) using linear probing.

```c
#define MAX 100

int HTsize;

struct Employee
{
    int key;
    int data;
};

struct Employee HT[MAX];

int hashFunction(int key)
{
    return key % HTsize;
}

void initHT(int size)
{
    HTsize = size;
    for (int i = 0; i < size; i++)
    {
        HT[i].key = -1;
        HT[i].data = 0;
    }
}

void insert(int key, int data)
{
    int index = hashFunction(key);
    while (HT[index].key != -1)
    {
        index = (index + 1) % HTsize;
    }
    HT[index].key = key;
```

```c
void insert(int key, int data)
{
    HT[index].data = data;
}

int search(int key)
{
    int index = hashFunction(key);
    while (HT[index].key != -1)
    {
        if (HT[index].key == key)
        {
            return index;
        }
        index = (index + 1) % HTsize;
    }
    return -1;
}

void printHT()
{
    for (int i = 0; i < HTsize; i++)
    {
        if (HT[i].key != -1)
        {
            printf("Index %d: Key %d, Data %d\n", i, HT[i].key, HT[i].data);
        }
        else
        {
            printf("Index %d: Empty\n", i);
        }
    }
}
```

```c
int main()
{
  int m = 10;
  initHT(m);

  insert(1234, 1001);
  insert(1232, 1002);
  insert(1357, 1003);
  insert(1345, 1004);
  insert(1234, 1005);

  printf("Hash Table contents:\n");
  printHT();

  int keyToSearch = 1357;
  int index = search(keyToSearch);
  if (index != -1)
  {
    printf("Found key %d at index %d with data %d\n", keyToSearch, index, HT[index].data);
  }
  else
  {
    printf("NOT FOUND.\n");
  }

  return 0;
}
```

Output:

```
Hash Table contents:
Index 0: Empty
Index 1: Empty
Index 2: Key 1232, Data 1002
Index 3: Empty
Index 4: Key 1234, Data 1001
Index 5: Key 1345, Data 1004
Index 6: Key 1234, Data 1005
Index 7: Key 1357, Data 1003
Index 8: Empty
Index 9: Empty
Found key 1357 at index 7 with data 1003
```