Date: 01/10/24

Program 1: **Array Operations**

Aim: Write a program to perform insertion, deletion, search and traversal in an array.

Algorithm:

Step 1: Define and Initialize Array

Define a global array `arr` of size `MAX` (100).

Initialize a variable `size` to `-1` to indicate the array is initially empty.

Step 2: Display the Operations Menu

Display the following options for the user:

1. Insert an element into the array.
2. Delete an element from the array.
3. Display all elements in the array.
4. Search for an element in the array.
5. Exit the program.

Take the user's choice as input.

Step 3: Perform Operation Based on Choice

Case 1: Insert an Element

- Ask the user for the element (`num`) to insert and the position (`pos`) where it should be inserted.
- Convert the position to zero-based indexing (`pos - 1`).
- Check if the position is valid (`0 <= pos <= size + 1`) and if the array is not full (`size < MAX - 1`):
- If invalid, display an error message.
- If valid:

Shift all elements from position `pos` to the right to create space.

Insert `num` at position `pos`.

Increment the size of the array by 1.

Display a success message.

Case 2: Delete an Element

- Ask the user for the position (`pos`) of the element to delete.
- Convert the position to zero-based indexing (`pos - 1`).
- Check if the position is valid (`0 <= pos <= size`) and if the array is not empty (`size >= 0`):
- If invalid or the array is empty, display an error message.
- If valid:

    Display the element being deleted (`arr[pos]`).

    Shift all elements from position `pos + 1` to the left.

    Decrement the size of the array by 1.

Case 3: Display Elements

- Check if the array is empty (`size < 0`):
- If the array is empty, display "Array is empty."
- If not, display all elements in the array from `arr[0]` to `arr[size]`.

Case 4: Search for an Element

- Ask the user for the element (`num`) to search for.
- Loop through the array from `arr[0]` to `arr[size]` and compare each element with `num`.
- If an element matches `num`, display "Element found at position `i + 1 `."
- If the element is not found by the end of the loop, display "Element not found in the array."

Case 5: Exit the Program

- Terminate the program.

Default Case: Invalid Input

- Display "Enter a valid choice!"

Step 4: Repeat the Process

- After performing the chosen operation, loop back to Step 2 until the user selects exit.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100

int arr[MAX];
int size = -1;

void insert();
void delete();
void display();
void search(); // New search function

void main()
{
    int choice;
    printf("\n---- Array Operations ----\n");

    while (1)
    {
        printf("\n-Operations-\n");
        printf("1. Insert\n2. Delete\n3. Display\n4. Search\n5. Exit\n");
        printf("Select an operation: ");
        scanf("%d", &choice);

        switch (choice)
        {
        case 1:
```

```
                insert();
                break;
            case 2:
                delete();
                break;
            case 3:
                display();
                break;
            case 4:
                search();
                break;
            case 5:
                exit(0);
            default:
                printf("Enter a valid choice!\n");
                break;
            }
        }
    }

    void insert()
    {
        int num, pos;
        printf("Enter element to be inserted: ");
        scanf("%d", &num);
        printf("Enter position: ");
        scanf("%d", &pos);

        pos--;
```

```
    if (pos < 0 || pos > size + 1 || size >= MAX - 1)
    {
        printf("Invalid position or array is full!\n");
        return;
    }

    for (int i = size + 1; i > pos; i--)
    {
        arr[i] = arr[i - 1];
    }

    arr[pos] = num;
    size++;
    printf("Element inserted successfully.\n");
}

void delete()
{
    int pos;
    printf("Enter position of element to delete: ");
    scanf("%d", &pos);

    pos--;

    if (size < 0)
    {
        printf("Array is empty!\n");
        return;
    }
    else if (pos < 0 || pos > size)
```

```
    {
        printf("Invalid position!\n");
        return;
    }

    printf("Element deleted is %d\n", arr[pos]);

    for (int i = pos; i < size; i++)
    {
        arr[i] = arr[i + 1];
    }

    size--;
}

void display()
{
    if (size < 0)
    {
        printf("Array is empty!\n");
        return;
    }
    printf("The elements are: ");
    for (int i = 0; i <= size; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

```
void search()
{
    int num, found = 0;
    printf("Enter element to search: ");
    scanf("%d", &num);

    for (int i = 0; i <= size; i++)
    {
        if (arr[i] == num)
        {
            printf("Element %d found at position %d.\n", num, i + 1);
            found = 1;
            break;
        }
    }
    if (!found)
    {
        printf("Element %d not found in the array.\n", num);
    }
}
```

Output:

```
---- Array Operations ----

-Operations-
1. Insert
2. Delete
3. Display
4. Exit
Select an operation: 1
Enter element to be inserted: 21
Enter position: 1
Element inserted successfully.

-Operations-
1. Insert
2. Delete
3. Display
4. Exit
Select an operation: 1
Enter element to be inserted: 4
Enter position: 2
Element inserted successfully.

-Operations-
1. Insert
2. Delete
3. Display
4. Exit
Select an operation: 1
Enter element to be inserted: 6
Enter position: 3
Element inserted successfully.

-Operations-
1. Insert
2. Delete
3. Display
4. Search
5. Exit
Select an operation: 2
Enter position of element to delete: 2
Element deleted is 4

-Operations-
1. Insert
2. Delete
3. Display
4. Search
5. Exit
Select an operation: 3
The elements are: 21 6

-Operations-
1. Insert
2. Delete
3. Display
4. Search
5. Exit
Select an operation: 4
Enter element to search: 6
Element 6 found at position 2.
```

Result: The program has executed successfully and required output is obtained.

Date: 01/10/24

Program 2: **Merge Arrays**

Aim: Write a program to merge two arrays.

Algorithm:

Step 1: Input Array Sizes

Step 2: Declare Arrays

Declare three arrays:

`a` of size `n` (to store the first array).

`b` of size `m` (to store the second array).

`c` of size `n + m` (to store the merged array).

Step 3: Input Array Elements

Input elements for the first array `a`:

For each index `i` from 0 to `n`, read the element `a[i]`.

Input elements for the second array `b`:

For each index `i` from 0 to `m`, read the element `b[i]`.

Step 4: Merge Arrays

Copy elements of array `a` into the first part of array `c`:

For each index `i` from 0 to `n`, set `c[i] = a[i]`.

Copy elements of array `b` into the remaining part of array `c`:

For each index `i` from 0 to `m-1`, set `c[n + i] = b[i]`

Step 4: Merge Arrays

For each index `i` from 0 to `n + m - 1`:

If `i < n` (index within the first array):

Copy the element from `a[i]` to `c[i]`.

Otherwise (index in the second array):

Copy the element from `b[i - n]` to `c[i]`.

Step 5: Output the Merged Array

Print all elements of the merged array `c`:

For each index `i` from 0 to `n + m - 1`, print `c[i]`.

Source Code:

```
#include <stdio.h>

void main()
{
    int n, m;
    printf("Enter the size of first array: ");
    scanf("%d", &n);
    printf("Enter the size of second array: ");
    scanf("%d", &m);

    int a[n], b[m], c[m + n];

    printf("\nEnter the elements of first array: \n");
    for (int i = 0; i < n; i++)
    {
        printf("a[%d] = ", i);
        scanf("%d", &a[i]);
    }
    printf("\nEnter the elements of second array: \n");
    for (int i = 0; i < m; i++)
    {
        printf("b[%d] = ", i);
        scanf("%d", &b[i]);
    }
    for (int i = 0; i < n + m; i++)
    {
```

```
            if (i < n)
            {
              c[i] = a[i];
            }
            else
            {
              c[i] = b[i - n];
            }
        }

        printf("\nThe merged array is: ");
        for (int i = 0; i < m + n; i++)
        {
            printf("%d ", c[i]);
        }
    }
```

Output:

```
Enter the elements of first array:
a[0] = 2
a[1] = 4
a[2] = 6
a[3] = 8
a[4] = 10

Enter the elements of second array:
b[0] = 3
b[1] = 6
b[2] = 9

The merged array is: 2 4 6 8 10 3 6 9
```

Result: The program has executed successfully and required output is obtained.

Date: 08/10/24

Program 3: **Linked List Operations**

Aim: Write a program to perform insertion, deletion,search and traversal in a linked list.

Algorithm:

Step1. Start

- Define a `struct node` that includes:
  - `data` (to store the value of the node).
  - `next` (a pointer to the next node).
- Initialize `head` as `NULL` to represent an empty linked list.

Step 2. Create the Linked List

Check if the list is empty (`head == NULL`).

- If the list is empty, prompt the user to enter the number of nodes (`n`).
- If `n` is 0, print a message and stop; otherwise, proceed.
- Take input for the data of the first node, allocate memory for it, and make it the `head`.
- Create subsequent nodes (`n-1` times):
  - Allocate memory for a new node.
  - Link the new node to the current node.
  - Assign data to the new node.

- If the list is not empty, print a message: "The list is already created."

Step 3. Traverse the List

Check if the list is empty (`head == NULL`).

- If the list is empty, print: "List is empty."
- Otherwise:
  - Use a temporary pointer (`temp`) to traverse the list.
  - For each node, print its `data` and move to the next node until `temp == NULL`.

Step 4. Insert Operations

Insert at Front

- Allocate memory for a new node.
- Prompt the user for data and assign it to the new node.
- Update the `next` pointer of the new node to point to the current `head`.
- Make the new node the `head`.

Insert at End

- Allocate memory for a new node.
- Prompt the user for data and assign it to the new node.
- Traverse to the last node of the list.
- Update the `next` pointer of the last node to point to the new node.
- Set the `next` of the new node to `NULL`.

Insert at a Specific Position

- Prompt the user for the position (`pos`) and data (`num`).
- Allocate memory for a new node and assign the data.
- Traverse the list until you reach the `(pos - 1)`th node.
- Update the `next` pointer of the new node to point to the current `next` node.
- Update the `next` pointer of the `(pos - 1)`th node to point to the new node.

Step 5. Delete Operations

Delete First Node

- Check if the list is empty (`head == NULL`).
- If empty, print: "List is empty."
- Otherwise:
  - Save the current `head` in a temporary pointer.
  - Update `head` to point to the second node (`head->next`).
  - Free the memory of the old head.

Delete Last Node

- Check if the list is empty (`head == NULL`).

- If empty, print: "List is empty."
- Otherwise:
  - Traverse the list to the second-to-last node.
  - Update its `next` pointer to `NULL`.
  - Free the memory of the last node.

Delete Node at a Specific Position

- Check if the list is empty (`head == NULL`).
  - If empty, print: "List is empty."
- Otherwise:
  - Prompt the user for the position (`pos`).
  - Traverse the list to the `(pos - 1)`th node.
  - Save the reference to the node at the `pos` in a temporary pointer.
  - Update the `next` pointer of the `(pos - 1)`th node to skip the node at `pos`.
  - Free the memory of the node at `pos`.

6. Search for an Element

Check if the list is empty (`head == NULL`).

- If empty, print: "The list is empty."

Otherwise:

- Prompt the user for the element to search (`key`).
- Traverse the list:
  - Compare the `data` of each node with `key`.
  - If a match is found, print: "Yes, `<key>` is present in the list," and stop.
- If the end of the list is reached without finding the `key`, print: "No, `<key>` is not present in the list."

Step 7. Main Function Logic

- Call `createList()` to initialize the linked list.
- Enter a loop:
  - Display a menu of operations:

1. Display the list.
2. Insert at the beginning.
3. Insert at the end.
4. Insert at a specific position.
5. Delete the first node.
6. Delete the last node.
7. Delete a node at a specific position.
8. Search for an element.
9. Exit the program.

- Prompt the user for their choice.
- Based on the choice, call the corresponding function to perform the operation.
- If the choice is `9`, exit the program.

Step 8. End

- The program terminates when user chooses choice 9.

<u>Source Code</u>:

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

struct node
{
    int data;
    struct node *next;
};
struct node *head = NULL;

void createList()
{
    if (head == NULL)
```

```
{
    int n;
    printf("\nEnter the number of nodes: ");
    scanf("%d", &n);
    if (n != 0)
    {
        int data;
        struct node *newnode;
        struct node *temp;
        newnode = malloc(sizeof(struct node));
        newnode->next = NULL;
        head = newnode;
        temp = head;
        printf("\nEnter number to be inserted : ");
        scanf("%d", &data);
        head->data = data;

        for (int i = 2; i <= n; i++)
        {
            newnode = malloc(sizeof(struct node));
            newnode->next = NULL;
            temp->next = newnode;
            printf("\nEnter number to be inserted : ");
            scanf("%d", &data);
            newnode->data = data;
            temp = temp->next;
        }
    }
    printf("\nThe list is created\n");
}
```

```
    else
        printf("\nThe list is already created\n");
}
void traverse()
{
    struct node *current;

    if (head == NULL)
        printf("\nList is empty\n");
    else
    {
        current = head;
        while (current != NULL)
        {
            printf("Data = %d\n", current->data);
            Current = current->next;
        }
    }
}
void insertAtFront()
{
    int num;
    struct node *newnode;
    newnode = malloc(sizeof(struct node));
    printf("\nEnter number to be inserted : ");
    scanf("%d", &num);

    newnode->data = num;
    newnode->next = head;
    head = newnode;
```

```
}
void insertAtEnd()
{
    int num;
    struct node *newnode, *current;
    newnode = malloc(sizeof(struct node));

    printf("\nEnter number to be inserted : ");
    scanf("%d", &num);

    newnode->next = NULL;
    newnode->data = num;
    current = head;
    while (current->next != NULL)
    {
        current = current->next;
    }
    current->next = newnode;
}
void insertAtPosition()
{
    struct node *current, *newnode;
    int pos, num, i = 1;
    newnode = malloc(sizeof(struct node));

    printf("\nEnter position: ");
    scanf("%d", &pos);
    printf("\nEnter data:");
    scanf("%d", &num);
```

```
        newnode->data = num;
        newnode->next = NULL;
        current = head;
        while (i < pos - 1)
        {
            current = current->next;
            i++;
        }
        newnode->next = current->next;
        current->next = newnode;
    }
    void deleteFront()
    {
        struct node *temp;
        if (head == NULL)
            printf("\nList is empty\n");
        else
        {
            temp = head;
            head = head->next;
            free(temp);
        }
    }

    void deleteEnd()
    {
        struct node *temp, *current;
        if (head == NULL)
            printf("\nList is Empty\n");
        else
```

```
    {
        current = head;
        while (current->next->next != NULL)
        {
            current = current->next;
        }
        temp = current->next;
        current->next = NULL;
        free(temp);
    }
}
void deletePosition()
{
    struct node *current, *temp;
    int i = 1, pos;

    if (head == NULL)
        printf("\nList is empty\n");
    else
    {
        printf("\nEnter index : ");

        scanf("%d", &pos);
        temp = malloc(sizeof(struct node));
        current = head;

        while (i < pos - 1)
        {
            current = current->next;
            i++;
```

```
        }
        temp = current->next;
        current->next = temp->next;
        free(temp);
    }
}

void search()
{
    int found = -1;
    struct node *current = head;

    if (head == NULL)
    {
        printf("nexted list is empty\n");
    }
    else
    {
        printf("\nEnter the element you want to search: ");
        int key;
        scanf("%d", &key);
        while (current != NULL)
        {
            if (current->data == key)
            {
                found = 1;
                break;
            }
            else
            {
```

```
                current = current->next;
            }
        }
        if (found == 1)
        {
            printf("Yes, %d is present in the list.\n", key);
        }
        else
        {
            printf("No, %d is not present in the list.\n", key);
        }
    }
}
int main()
{
    createList();
    int choice;
    while (1)
    {
        printf("\n1. To display list\n");
        printf("2. For insertion at heading\n");
        printf("3. For insertion at end\n");
        printf("4. For insertion at any position\n");
        printf("5. For deletion of first element\n");
        printf("6. For deletion of last element\n");
        printf("7. For deletion of element at any position\n");
        printf("8. Search an element=\n");
        printf("9. To exit\n");
        printf("\nEnter Choice :\n");
        scanf("%d", &choice);
```

```
switch (choice)
{
case 1:
    traverse();
    break;
case 2:
    insertAtFront();
    break;
case 3:
    insertAtEnd();
    break;
case 4:
    insertAtPosition();
    break;
case 5:
    deleteFront();
    break;
case 6:
    deleteEnd();
    break;
case 7:
    deletePosition();
    break;
case 8:
    search();
    break;
case 9:
    exit(1);
    break;
default:
```

```
                printf("Incorrect Choice\n");
            }
        }
        return 0;
    }
```

Output:

```
Enter the number of nodes: 4

Enter number to be inserted : 3

Enter number to be inserted : 5

Enter number to be inserted : 7

Enter number to be inserted : 9

The list is created

        1 To display list
        2 For insertion at heading
        3 For insertion at end
        4 For insertion at any position
        5 For deletion of first element
        6 For deletion of last element
        7 For deletion of element at any position
        12 Search an element=
        13 To exit

Enter Choice :
2

Enter number to be inserted : 1

        1 To display list
        2 For insertion at heading
        3 For insertion at end
        4 For insertion at any position
        5 For deletion of first element
        6 For deletion of last element
        7 For deletion of element at any position
        12 Search an element=
        13 To exit

Enter Choice :
3

Enter number to be inserted : 10

        1 To display list
        2 For insertion at heading
        3 For insertion at end
        4 For insertion at any position
        5 For deletion of first element
        6 For deletion of last element
        7 For deletion of element at any position
        12 Search an element=
        13 To exit

Enter Choice :
4

Enter position: 2
```

```
Enter data:6

        1 To display list
        2 For insertion at heading
        3 For insertion at end
        4 For insertion at any position
        5 For deletion of first element
        6 For deletion of last element
        7 For deletion of element at any position
        12 Search an element=
        13 To exit

Enter Choice :
1
Data = 1
Data = 6
Data = 3
Data = 5
Data = 7
Data = 9
Data = 10

        1 To display list
        2 For insertion at heading
        3 For insertion at end
        4 For insertion at any position
        5 For deletion of first element
        6 For deletion of last element
        7 For deletion of element at any position
        12 Search an element=
        13 To exit

Enter Choice :
5

        1 To display list
        2 For insertion at heading
        3 For insertion at end
        4 For insertion at any position
        5 For deletion of first element
        6 For deletion of last element
        7 For deletion of element at any position
        12 Search an element=
        13 To exit

Enter Choice :
6

        1 To display list
        2 For insertion at heading
        3 For insertion at end
        4 For insertion at any position
        5 For deletion of first element
        6 For deletion of last element
        7 For deletion of element at any position
        12 Search an element=
```

```
Enter Choice :
7

Enter index : 3

        1 To display list
        2 For insertion at heading
        3 For insertion at end
        4 For insertion at any position
        5 For deletion of first element
        6 For deletion of last element
        7 For deletion of element at any position
        12 Search an element=
        13 To exit

Enter Choice :
1
Data = 6
Data = 3
Data = 7
Data = 9

        1 To display list
        2 For insertion at heading
        3 For insertion at end
        4 For insertion at any position
        5 For deletion of first element
        6 For deletion of last element
        7 For deletion of element at any position
        12 Search an element=
        13 To exit

Enter Choice :
12

Enter the element you want to search: 9
Yes, 9 is present in the list.

        1 To display list
        2 For insertion at heading
        3 For insertion at end
        4 For insertion at any position
        5 For deletion of first element
        6 For deletion of last element
        7 For deletion of element at any position
        12 Search an element=
        13 To exit

Enter Choice :
13
```

Result: The program has executed successfully and required output is obtained.

Date: 08/10/24

Program 4: **Stack using Linked list**

Aim: Write a program to implement stack using linked list.

Algorithm:

Step 1. Initialize

- Define a global pointer `top` to `NULL`. This pointer will always point to the top of the stack.

Step 2. Push Operation

Input: An integer `value`.

1. Create a new node (`newNode`) using dynamic memory allocation.
2. Check if memory allocation is successful:
   - If not, print "Memory overflow!" and exit the function.
3. Assign the value to `newNode->data`.
4. Set `newNode->next` to point to the current `top`.
5. Update `top` to point to `newNode`.
6. End the function.

Step 3. Pop Operation

1. Check if the stack is empty (`top == NULL`):
   - If true, print "Stack underflow!" and return an error code (e.g., `-1`).
2. Create a temporary pointer `deleteNode` to point to `top`.
3. Save the data from the top node (`deleteNode->data`) into a variable (`element`).
4. Update `top` to point to the next node (`top->next`).
5. Free the memory allocated for `deleteNode`.
6. Return the value stored in `element`.

Step 4. Display Operation

1. Check if the stack is empty (`top == NULL`):
   - If true, print "Stack is empty!" and exit the function.

2. Create a temporary pointer `current` to point to `top`.
3. Traverse the stack using `current`:
   - Print `current->data`.
   - Move `current` to the next node (`current = current->next`).
4. End the function.

Step 5. Main Program

1. Display a menu to the user with the following options:
   - Push
   - Pop
   - Display
   - Exit
2. Initialize a variable `choice` to store the user's selection.
3. Use a loop to repeatedly display the menu until the user chooses "Exit".
4. Depending on the value of `choice`:
   - 1 (Push): Prompt the user for a value, then call the `push` function with this value.
   - 2 (Pop): Check if the stack is empty. If not, call the `pop` function and display the popped element.
   - 3 (Display): Call the `display` function to show the stack contents.
   - 4 (Exit): Print an exit message and terminate the program.
   - Invalid Option: Print an error message and prompt the user again.

Step 6. End the program.

Source Code:

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

```
} *top = NULL;

void push(int value);
int pop();
void display();

int main()
{
    printf("\n----STACK USING LINKED LIST----");

    int choice = 0;

    while (choice != 4)
    {
        printf("\nOperations\n");
        printf("1. Push\n2. Pop\n3. Display\n4. Exit\n\nSelect an operation: 
");
        scanf("%d", &choice);
        switch (choice)
        {
        case 1:
        {
            int num;
            printf("Enter element to push: ");
            scanf("%d", &num);
            push(num);
            break;
        }
        case 2:
            if (top == NULL)
```

```
            {
                printf("Stack is empty!\n");
            }
            else
            {
                int num = pop();
                printf("The popped element is: %d\n", num);
            }
            break;
        case 3:
            display();
            break;
        case 4:
            printf("Exiting program. Goodbye!\n");
            exit(0);
            break;
        default:
            printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}
void push(int value)
{
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));

    if (newNode == NULL)
    {
```

```
            printf("Memory overflow!\n");
        }
        else
        {
            newNode->data = value;
            newNode->next = top;
            top = newNode;
        }
    }
    int pop()
    {
        struct node *deleteNode = top;
        int element = deleteNode->data;
        top = top->next;
        free(deleteNode);

        return element;
    }

    void display()
    {
        if (top == NULL)
        {
            printf("Stack is empty!\n");
        }
        else
        {
            struct node *current = top;
            printf("Stack elements: ");
            while (current != NULL)
```

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

Output:

```
----STACK USING LINKED LIST----
Operations
1. Push
2. Pop
3. Display
4. Exit

Select an operation: 1
Enter element to push: 7

Operations
1. Push
2. Pop
3. Display
4. Exit

Select an operation: 1
Enter element to push: 8

Operations
1. Push
2. Pop
3. Display
4. Exit

Select an operation: 3
Stack elements: 8  7

Operations
1. Push
2. Pop
3. Display
4. Exit

Select an operation: 2
The popped element is: 8
```

```
Operations
1. Push
2. Pop
3. Display
4. Exit

Select an operation: 2
The popped element is: 7

Operations
1. Push
2. Pop
3. Display
4. Exit

Select an operation: 2
Stack is empty!

Operations
1. Push
2. Pop
3. Display
4. Exit

Select an operation: 4
Exiting program. Goodbye!
```

Result: The program has executed successfully and required output is obtained.

Date: 15/10/24

Program 5: **Queue using Linked List**

Aim: Write a program implement queue using linked list.

Algorithm:

Step 1: Start

- Define a `struct node` with:
  - `data` (to store the value of the node).
  - `next` (a pointer to the next node).
- Initialize `front` and `rear` as `NULL` to represent an empty queue.

Step 2: Enqueue Operation (Insert at the Rear)

1. Allocate Memory for a new node using `malloc`.
2. Take Input for the data to be inserted into the queue.
3. Assign Data to the new node (`newnode->data`).
4. Set Next Pointer of the new node to `NULL`.
5. Check if the Queue is Empty (`rear == NULL`):
   - If the queue is empty (both `front` and `rear` are `NULL`):
     - Set both `front` and `rear` to the new node (new node becomes both front and rear).
   - If the queue is not empty:
     - Set the `next` pointer of the current `rear` node to the new node.
     - Update `rear` to point to the new node.

Step 3: Dequeue Operation (Remove from the Front)

1. Check if the Queue is Empty (`front == NULL`):
   - If empty, print: "Queue Empty!!" and return.
2. Save the Front Node in a temporary pointer (`temp`).
3. Print the Deleted Element (`front->data`).
4. Move Front Pointer:
   - Set `front` to point to the next node (`front = front->next`).

Step 4: Display the Queue

1. Check if the Queue is Empty (`front == NULL`):
   - If empty, print: "Queue Empty".
2. Traverse the Queue:
   - Use a temporary pointer (`current`) to traverse from `front` to `rear`.
   - For each node, print the `data` and move to the next node until `current = NULL`.
3. End Display Operation and return to the menu.

Step 5: Main Function Logic

1. Start the Queue Program by initializing `front` and `rear` as `NULL`.
2. Enter a Loop to repeatedly display the menu and handle user input:
   - Display the menu with options:
     1. Enqueue (Add element to the queue).
     2. Dequeue (Remove element from the queue).
     3. Display the Queue.
     4. Exit the program.
   - Prompt the user to enter their choice and process accordingly:
     - If the choice is `1` (Enqueue), call the `enqueue()` function.
     - If the choice is `2` (Dequeue), call the `dequeue()` function.
     - If the choice is `3` (Display), call the `display()` function.
     - If the choice is `4` (Exit), set `is_running = 0` to exit the loop.
     - If the choice is invalid, print: "Wrong Choice :(".
3. End the Program when the user selects "Exit".

Step 6: End

- The program terminates when the user selects option `4` to exit.

<u>Source Code</u>:

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

void enqueue();
void dequeue();
void display();

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

node *front = NULL;
node *rear = NULL;

int main()
{
    int is_running = 1, ch;
    while (is_running)
    {
        printf("\n----Queue Using Linked List----\n");
        printf("\n\n1. Enqueue\n2. Dequeue\n3. Display\n4. Exit\nEnter your
        choice: ");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
            enqueue();
            break;
```

```
        case 2:
            dequeue();
            break;
        case 3:
            display();
            break;
        case 4:
            is_running = 0;
            break;
        default:
            printf("\nWrong Choice :(\n");
            break;
        }
    }
    return 0;
}
void enqueue()
{
    node *newnode;
    newnode = (node *)malloc(sizeof(node));

    if (newnode == NULL)  // Check if memory allocation was successful
    {
        printf("Memory allocation failed!\n");
        return;
    }

    int data;
    printf("\nEnter the data to be added: ");
    scanf("%d", &data);
```

```
    newnode->data = data;
    newnode->next = NULL;
    if (rear == NULL && front == NULL)
    {
        rear = newnode;
        front = newnode;
    }
    else
    {
        rear->next = newnode;
        rear = newnode;
    }
}
void dequeue()
{
    if (front == NULL)
    {
        printf("\nQueue Empty!!\n");
    }
    else
    {
        node *temp;
        printf("The deleted element is %d\n", front->data);
        temp = front;
        front = front->next;
        if (front == NULL)
        {
            rear = NULL;
        }
        free(temp);
```

```
    }
}
void display()
{
    if (front == NULL)
    {
        printf("\nQueue Empty\n");
        return;
    }
    else
    {
        node *current = front;

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

Output:

```
----Queue Using Linked List----

1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1

Enter the data to be added: 23

----Queue Using Linked List----

1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1

Enter the data to be added: 54

----Queue Using Linked List----

1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1

Enter the data to be added: 4

----Queue Using Linked List----

1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3
23  54  4
```

```
----Queue Using Linked List----

1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2
The deleted element is 23

----Queue Using Linked List----

1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2
The deleted element is 54

----Queue Using Linked List----

1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2
The deleted element is 4

----Queue Using Linked List----

1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2

Queue Empty!!
```

Result: The program has executed successfully and required output is obtained.

Date: 15/10/24

Program 6: **Doubly Linked List**

Aim: Write a program to perform operations on doubly linked list.

Algorithm:

Step 1. Create Function

1. Prompt the user to input the number of nodes (`n`).
2. Loop `n` times to create `n` nodes:
   - For each iteration:
     - Allocate memory for a new node (`newNode`).
     - If memory allocation fails, print an error and return.
     - Prompt the user to input the data for the new node.
     - Set the `next` and `prev` pointers of `newNode` to `NULL`.
     - If the list is empty (i.e., `head` is `NULL`):
       - Set `head` and `tail` to point to `newNode`.
     - Otherwise:
       - Set `tail->next` to point to `newNode` and `newNode->prev` to point to `tail`.
       - Update `tail` to `newNode`.

2. Display Function

1. Prompt the user to choose how to display the list:
   - Option 1: Display from the start.
   - Option 2: Display from the end.
2. If the user chooses 1 (start):
   - Start from `head` and traverse the list by moving from one node to the next (`temp = temp->next`) until reaching the end (`temp = NULL`).
   - Print the data of each node during traversal.
3. If the user chooses 2 (end):
   - Start from `tail` and traverse the list by moving from one node to the previous (`temp = temp->prev`) until reaching the beginning (`temp = NULL`).

- Print the data of each node during traversal.

4. If the choice is invalid, print an error message.

3. Insert Function

1. Prompt the user to choose where to insert the node:
   - Option 1: Insert at the top (beginning).
   - Option 2: Insert at the bottom (end).
   - Option 3: Insert at a specific position.
2. For each choice, perform the following:
   - Top Insertion:
     - Create a new node and set its `next` to `head`.
     - If `head` is not `NULL`, set `head->prev` to the new node.
     - Set `head` to point to the new node.
     - If the list was empty (i.e., `tail` is `NULL`), also set `tail` to the new node.
   - Bottom Insertion:
     - Create a new node and set `tail->next` to point to the new node.
     - Set `newNode->prev` to `tail`.
     - Update `tail` to point to `newNode`.
     - If the list was empty (i.e., `head` is `NULL`), set both `head` and `tail` to `newNode`.
   - Specific Position Insertion:
     - Prompt the user to enter the position for insertion.
     - Traverse the list until reaching the desired position (`temp`).
     - Insert the new node in the middle by updating the `next` and `prev` pointers of surrounding nodes.
     - If the position is out of range, print an error.

4. Delete Function

1. Prompt the user to choose where to delete a node:
   - Option 1: Delete from the top (beginning).
   - Option 2: Delete from the bottom (end).
   - Option 3: Delete from a specific position.

2. For each choice, perform the following:

   - Top Deletion:

     - If the list is empty (i.e., `head == NULL`), print an error message.

     - Set `head` to `head->next`. If the new `head` is not `NULL`, set `head->prev` to `NULL`.

     - If `head` becomes `NULL` (empty list), set `tail` to `NULL`.

     - Free the memory for the deleted node.

   - Bottom Deletion:

     - If the list is empty (i.e., `tail == NULL`), print an error message.

     - Set `tail` to `tail->prev`. If the new `tail` is not `NULL`, set `tail->next` to `NULL`.

     - If `tail` becomes `NULL` (empty list), set `head` to `NULL`.

     - Free the memory for the deleted node.

   - Specific Position Deletion:

     - Prompt the user to enter the position for deletion.

     - Traverse the list until reaching the desired position (`temp`).

     - Adjust the `next` and `prev` pointers of surrounding nodes to bypass the node to be deleted.

     - If the node is at the beginning (`head`), update `head`.

     - If the node is at the end (`tail`), update `tail`.

     - Free the memory for the deleted node.

5. Main Function

   1. Print a menu with options for the user to choose:

      - 1: Create a doubly linked list.

      - 2: Display the list.

      - 3: Insert a node.

      - 4: Delete a node.

      - 5: Exit the program.

   2. Continuously prompt the user for their choice:

      - Call the respective function based on the user's choice.

      - If the choice is invalid, print an error message.

   3. If the user chooses 5 (exit), terminate the program.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node
{
    int data;
    struct Node *next;
    struct Node *prev;
} Node;

Node *head = NULL, *tail = NULL, *newNode, *temp;

void create()
{
    int n;
    printf("Enter Number of Nodes: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
    {
        printf("Enter Data: ");
        newNode = (Node *)malloc(sizeof(Node));
        if (newNode == NULL)
        {
            printf("Memory allocation failed!\n");
            return;
        }
        scanf("%d", &newNode->data);
        newNode->next = NULL;
        newNode->prev = NULL;
        if (head == NULL)
```

```
        {
            head = newNode;
            tail = newNode;
        }
        else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
    }
}
void display()
{
    int choice;
    printf("Display From\n1. Start\n2. End\nChoose: ");
    scanf("%d", &choice);
    if (choice == 1)
    {
        for (temp = head; temp != NULL; temp = temp->next)
        {
            printf("%d\t", temp->data);
        }
    }
    else if (choice == 2)
    {
        for (temp = tail; temp != NULL; temp = temp->prev)
        {
            printf("%d\t", temp->data);
        }
    }
```

```
    else{
        printf("Invalid choice!\n");
    }
    printf("\n");
}
void insert()
{
    int choice, position;
    printf("Data to be inserted\n1. At the Top\n2. At the Bottom\n3. In
Between\nChoose(1/2/3): ");
    scanf("%d", &choice);
    newNode = (Node *)malloc(sizeof(Node));
    if (newNode == NULL)
    {
        printf("Memory allocation failed!\n");
        return;
    }
    printf("Enter The Data: ");
    scanf("%d", &newNode->data);
    newNode->next = NULL;
    newNode->prev = NULL;

    if (choice == 1)
    {
        newNode->next = head;
        if (head != NULL)
            head->prev = newNode;
        head = newNode;
        if (tail == NULL)  // If list was empty
            tail = newNode;
```

```
    }
    else if (choice == 2){
        if (tail != NULL)
        {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
        else{
            head = newNode;
            tail = newNode;
        }
    }
    else{
        printf("Enter the position: ");
        scanf("%d", &position);
        if (position < 1)
        {
            printf("Invalid position!\n");
            return;
        }
        temp = head;
        for (int i = 1; temp != NULL && i < position; i++)
        {
            temp = temp->next;
        }
        if (temp == NULL)
        {
            printf("Position out of range!\n");
            return;
```

```
        }
        newNode->next = temp->next;
        if (temp->next != NULL)
            temp->next->prev = newNode;
        temp->next = newNode;
        newNode->prev = temp;
    }
}
void delete()
{
    int choice, position;
    printf("Data to be Deleted\n1. From the Top\n2. From the Bottom\n3. In
Between\nChoose(1/2/3): ");
    scanf("%d", &choice);
    if (choice == 1)
    {
        if (head == NULL)
        {
            printf("List is empty!\n");
            return;
        }
        temp = head;
        head = head->next;
        if (head != NULL)
            head->prev = NULL;
        if (head == NULL)
            tail = NULL;
    }
    else if (choice == 2)
    {
```

```
        if (tail == NULL)
        {
            printf("List is empty!\n");
            return;
        }
        temp = tail;
        tail = tail->prev;
        if (tail != NULL)
            tail->next = NULL;
        if (tail == NULL)  //
            head = NULL;
    }
    else
    {
        printf("Enter the position: ");
        scanf("%d", &position);
        if (position < 1 || head == NULL)
        {
            printf("Invalid position or empty list!\n");
            return;
        }
        temp = head;
        for (int i = 1; temp != NULL && i < position; i++)
        {
            temp = temp->next;
        }
        if (temp == NULL)
        {
            printf("Position out of range!\n");
            return;
```

```
        }
        if (temp->prev != NULL)
            temp->prev->next = temp->next;
        if (temp->next != NULL)
            temp->next->prev = temp->prev;
        if (temp == head)
            head = temp->next;
        if (temp == tail)
            tail = temp->prev;
        free(temp);
    }
}
void main()
{
    int choice;
    while (1)
    {
        printf("\n1. Create a LL\n2. Display the LL\n3. Insert Elements
into the LL\n4. Delete\n5. End\nChoose: ");
        scanf("%d", &choice);
        switch (choice)
        {
        case 1:
            create();
            break;
        case 2:
            display();
            break;
        case 3:
            insert();
```

```
            break;
        case 4:
            delete();
            break;
        case 5:
            exit(0);
        default:
            printf("Wrong choice.\n");
            break;
        }
    }
}
```

Output:

```
1. Create a LL
2. Display the LL
3. Insert Elements into the LL
4. Delete
5. End
Choose: 1
Enter Number of Nodes: 3
Enter Data: 5
Enter Data: 7
Enter Data: 9

1. Create a LL
2. Display the LL
3. Insert Elements into the LL
4. Delete
5. End
Choose: 3
Data to be inserted
1. At the Top
2. At the Bottom
3. In Between
Choose(1/2/3): 1
Enter The Data: 10

1. Create a LL
2. Display the LL
3. Insert Elements into the LL
4. Delete
5. End
Choose: 2
Display From
1. Start
2. End
Choose: 1
10      5       7       9

1. Create a LL
2. Display the LL
3. Insert Elements into the LL
4. Delete
5. End
Choose: 4
Data to be Deleted
1. From the Top
2. From the Bottom
3. In Between
Choose(1/2/3): 3
Enter the position: 2

1. Create a LL
2. Display the LL
3. Insert Elements into the LL
4. Delete
5. End
Choose: 2
Display From
1. Start
2. End
Choose: 1
10      7       9

1. Create a LL
2. Display the LL
3. Insert Elements into the LL
4. Delete
5. End
Choose: 5
```

Result: The program has executed successfully and required output is obtained.

Date: 22/10/24

Program 7: **Circular Linked List**

Aim: Write a program to perform operations on circular linked list.

Algorithm:

Step 1. Start

Step 2. Main Menu (Loop)

Continuously prompt the user for an operation until the user decides to exit:

- 1: Insert at beginning
- 2: Insert at end
- 3: Insert at a specific position
- 4: Delete a node at a specific position
- 5: Display the list
- 6: Exit

Step 3. Insert at Beginning (insatbeg)

- Create a new node.
- Read data from the user.
- If the list is empty:
  - Point the new node to itself (as it's the only node).
- If the list is not empty:
  - Traverse the list to find the last node (which points to the head).
  - Set the new node's next to point to the current head.
  - Update the head to the new node.
  - Update the last node’s next to point to the new head (maintaining the circular link).

Step 4. Insert at End (insatend)

- Create a new node.
- Read data from the user.
- If the list is empty:

- Point the new node to itself (circular link).

- If the list is not empty:

  - Traverse the list to find the last node (which points to the head).

  - Set the last node's next to the new node.

  - Set the new node's next to the head.

5. Insert at Position (insatpos)

- Create a new node.

- Read data and position from the user.

- If the list is empty or inserting at the first position:

  - Call `insatbeg()` to insert at the beginning.

- Otherwise:

  - Traverse the list to find the node just before the desired position.

  - Insert the new node after the current node and update the next pointers.

Step 6. Delete a Node (delete)

- Read position to delete from the user.

- If the list is empty, print an error message and exit.

- If the position is 1:

  - If there's only one node, set `head` to `NULL`.

  - Otherwise, find the last node, update its next pointer to the second node, and free the current head node.

- Otherwise:

  - Traverse the list to find the node at the given position.

  - Update the previous node's next pointer to skip the node being deleted and free the node.

Step 7. Display the List (display)

- If the list is empty, print a message indicating that there are no nodes to display.

- Otherwise:

  - Traverse the list and print the data of each node until you reach the head again.

Step 8. Exit

- The user exits the program, and the loop ends.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
void insatbeg();
void insatend();
void insatpos();
void display();
void delete();

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

node *head = NULL;

void main()
{
    int is_running = 1, ch;
    while (is_running)
    {
        printf("\n1.Insert at beginning\n2.Insert at end\n3.Insert at 
position\n4.Delete\n5.Display\n6.Exit\nEnter your choice: ");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
```

```
                insatbeg();
                break;
            case 2:
                insatend();
                break;
            case 3:
                insatpos();
                break;
            case 4:
                delete();
                break;
            case 5:
                display();
                break;
            case 6:
                is_running = 0;
                break;
            default:
                printf("\nWrong Choice :(\n");
                break;
            }
        }
    }
    void insatbeg()
    {
        node *newnode;
        newnode = (node *)malloc(sizeof(node));
        int data;
        printf("\nEnter the data to be added: ");
        scanf("%d", &data);
```

```
    newnode->data = data;
    if (head == NULL)
    {
        head = newnode;
        newnode->next = head;
    }
    else
    {
        node *current = head;
        while (current->next != head)
        {
            current = current->next;
        }
        newnode->next = head;
        head = newnode;
        current->next = head;
    }
}
void insatend()
{
    node *newnode;
    newnode = (node *)malloc(sizeof(node));
    int data;
    printf("\nEnter the data to be added: ");
    scanf("%d", &data);
    newnode->data = data;
    newnode->next = NULL;

    if (head == NULL)
    {
```

```
        head = newnode;
        newnode->next = head;
    }
    else
    {
        node *current = head;
        while (current->next != head)
        {
            current = current->next;
        }
        current->next = newnode;
        newnode->next = head;
    }
}
void insatpos()
{
    node *newnode;
    newnode = (node *)malloc(sizeof(node));
    int data, pos;
    printf("\nEnter the data to be added: ");
    scanf("%d", &data);
    newnode->data = data;
    printf("\nEnter the position to insert the data: ");
    scanf("%d", &pos);

    if (head == NULL || pos == 1)
    {
        insatbeg();
        return;
    }
```

```
    node *current = head;
    for (int i = 1; current->next != head && i < pos - 1; i++)
    {
        current = current->next;
    }

    newnode->next = current->next;
    current->next = newnode;
}
void display()
{
    if (head == NULL)
    {
        printf("\n!!There are no nodes to display!!\n");
        return;
    }
    node *current = head;
    do
    {
        printf("%d ", current->data);
        current = current->next;
    } while (current != head);
    printf("\n");
}
void delete()
{
    if (head == NULL) {
        printf("\n!!There are no nodes to delete!!\n");
        return;
```

```
    }
    int pos;
    printf("\nEnter the position to delete: ");
    scanf("%d", &pos);
    node *current = head;
    if (pos == 1)
    {
        if (current->next == head)
        {
            free(current);
            head = NULL;
        }
        else
        {
            while (current->next != head)
            {
                current = current->next;
            }
            node *temp = head;
            head = head->next;
            current->next = head;
            free(temp);
        }
    }
    else
    {
        for (int i = 1; current->next != head && i < pos - 1; i++)
        {
            current = current->next;
        }
```

```
        if (current->next == head)
        {
            printf("\nPosition exceeds list size.\n");
        }
        else
        {
            node *temp = current->next;
            current->next = current->next->next;
            free(temp);
        }
    }
}
```

Output:

```
1.Insert at beginning
2.Insert at end
3.Insert at position
4.Delete
5.Display
6.Exit
Enter your choice: 1

Enter the data to be added: 5

1.Insert at beginning
2.Insert at end
3.Insert at position
4.Delete
5.Display
6.Exit
Enter your choice: 2

Enter the data to be added: 6

1.Insert at beginning
2.Insert at end
3.Insert at position
4.Delete
5.Display
6.Exit
Enter your choice: 3
```

```
Enter the data to be added: 9

Enter the position to insert the data: 2

1.Insert at beginning
2.Insert at end
3.Insert at position
4.Delete
5.Display
6.Exit
Enter your choice: 4

Enter the position to delete: 3

1.Insert at beginning
2.Insert at end
3.Insert at position
4.Delete
5.Display
6.Exit
Enter your choice: 5
5 9

1.Insert at beginning
2.Insert at end
3.Insert at position
4.Delete
5.Display
6.Exit
Enter your choice: 6
```

Result: The program has executed successfully and required output is obtained.

Date: 22/10/24

Program 8: **Set using Bit vector**

Aim: Write a program to implement set using bit vector.

Algorithm:

Step 1. Input the size of the universal set:

- Read the value `n` representing the size of the universal set.
- Read the `n` elements of the universal set `U`.

Step 2. Input the first set

- Read the size `n1` of the first set.
- Ensure `n1` is valid (it should be less than or equal to `n` and greater than 0).
- Read `n1` elements for the first set `set1` from the universal set `U` (elements must be from `U` and not repeated).

Step 3. Input the second set

- Read the size `n2` of the second set.
- Ensure `n2` is valid (it should be less than or equal to `n` and greater than 0).
- Read `n2` elements for the second set `set2` from the universal set `U` (elements must be from `U` and not repeated).

Step 4. Create Bit Vectors for the sets

- Create two bit vectors `S1` and `S2` of size `n` (one for each set).
- Initialize `S1[i]` to 1 if the `i-th` element of the universal set is in `set1`, otherwise set it to 0.
- Similarly, initialize `S2[i]` to 1 if the `i-th` element of the universal set is in `set2`, otherwise set it to 0.

Step 5. Compute the Union of sets (S1 ∪ S2):

- Create a new bit vector `setUnion` of size `n`.
- For each index `i`, set `setUnion[i]` to 1 if either `S1[i] == 1` or `S2[i] == 1`, otherwise set it to 0.

Step 6. Compute the Intersection of sets (S1 ∩ S2):

- Create a new bit vector `setIntersection` of size `n`.
- For each index `i`, set `setIntersection[i]` to 1 if both `S1[i] == 1` and `S2[i] == 1`, otherwise set it to 0.

Step 7. Print the results:

- Print the Universal set `U`, the bit vector `S1` for set 1, the bit vector `S2` for set 2, the bit vector `setUnion` for the union of the sets, and the bit vector `setIntersection` for the intersection of the sets.

Source Code:

```
#include <stdio.h>


int isPresent(int arr[], int size, int value);

void readSet(int set[], int universal[], int size);

void printSet(int set[], int size);

int n, n1, n2;


void main()

{

  printf("Enter the size of the universal set: ");

  scanf("%d", &n);

  int U[n];

  printf("Enter the elements in the Universal Set: \n");

  readSet(U, NULL, n);


  printf("Enter the size of set 1: ");

  scanf("%d", &n1);

  if (n1 > n || n1 < 0) {

        printf("Invalid size!\n");

        return;

  }

  int set1[n1];

  printf("Enter the elements in set 1: \n");

  readSet(set1, U, n1);

  printf("Enter the size of set 2: ");
```

```
    scanf("%d", &n2);
    if (n2 > n || n2 < 0) {
        printf("Invalid size!\n");
        return;
    }
    int set2[n2];
    printf("Enter the elements in set 2: \n");
    readSet(set2, U, n2);

    int S1[n], S2[n];
    for (int i = 0; i < n; i++) {
        S1[i] = 0, S2[i] = 0;
        for (int j = 0; j < n1; j++) {
            if (U[i] == set1[j]) {
                S1[i] = 1;
                break;
            }
        }
        for (int j = 0; j < n2; j++) {
            if (U[i] == set2[j]) {
                S2[i] = 1;
                break;
            }
        }
    }
    int setUnion[n];
    for (int i = 0; i < n; i++) {
        if (S1[i] == 1 || S2[i] == 1)
            setUnion[i] = 1;
        else
```

```
            setUnion[i] = 0;
    }
    int setIntersection[n];
    for (int i = 0; i < n; i++) {
        if (S1[i] == 1 && S2[i] == 1)
            setIntersection[i] = 1;
        else
            setIntersection[i] = 0;
    }
    printf("U    : ");
    printSet(U, n);
    printf("Set 1: ");
    printSet(S1, n);
    printf("Set 2: ");
    printSet(S2, n);
    printf("S1uS2: ");
    printSet(setUnion, n);
    printf("S1nS2: ");
    printSet(setIntersection, n);
}
int isPresent(int arr[], int size, int value) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == value)
            return 1;
    }
    return 0;
}
void readSet(int set[], int universal[], int size) {
    int element;
    for (int i = 0; i < size; i++) {
```

```
        printf("Element %d: ", i + 1);
        scanf("%d", &element);

        if (!isPresent(set, i, element) && (universal == NULL ||
isPresent(universal, n, element))) {
            set[i] = element;
        } else {
            printf("Invalid Entry!\n");
            i--;
        }
    }
}
void printSet(int set[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", set[i]);
    }
    printf("\n");
}
```

Output:

```
Enter the size of the universal set: 6
Enter the elements in the Universal Set:
Element 1: 2
Element 2: 4
Element 3: 6
Element 4: 8
Element 5: 10
Element 6: 12
Enter the size of set 1: 3
Enter the elements in set 1:
Element 1: 3
Invalid Entry!
Element 1: 2
Element 2: 6
Element 3: 8
Enter the size of set 2: 2
Enter the elements in set 2:
Element 1: 10
Element 2: 2
U    : 2 4 6 8 10 12
Set 1: 1 0 1 1 0 0
Set 2: 1 0 0 0 1 0
S1uS2: 1 0 1 1 1 0
S1nS2: 1 0 0 0 0 0
```

Result: The program has executed successfully and required output is obtained.

Date: 29/10/24

Program 9: **Binary Search Tree**

Aim: Write a program to implement binary search tree.

Algorithm:

Main Function

1. Initialize the program:
   - Set `root` to `NULL`.
   - Set `is_running` to `1`.
2. Display the menu:
   - Option 1: Add a node to the tree by calling `add_node()`.
   - Option 2: Traverse the tree by calling `traverse(root)`.
   - Option 3: Delete a node by calling `delete_node(root, element)` with the user-specified `element`.
   - Option 4: Search for a node by calling `search(key, root)` with the user-specified `key`.
   - Option 5: Exit the program.
3. Repeat until the user exits (using `is_running = 0` or selecting Exit).

Function: add_node()

1. Allocate memory for a new node. If allocation fails, display an error and return.
2. Read the `data` to be added to the tree.
3. If the tree is empty (`root == NULL`), set the new node as the root.
4. Otherwise:
   - Traverse the tree to find the correct position for the new node.
   - Insert the node as a left or right child, depending on the value.
   - If the value already exists, display a message and free the allocated node.

Function: `traverse(root)`

1. Display traversal options:
   - Inorder traversal: Call `inorder(root)`.
   - Preorder traversal: Call `preorder(root)`.

- Postorder traversal: Call `postorder(root)`.

2. Perform the selected traversal and display the result.

Function: `inorder(root)`

1. If the tree is not empty:
   - Recursively call `inorder(root->left)`.
   - Display the root's data.
   - Recursively call `inorder(root->right)`.

Function: `preorder(root)`

1. If the tree is not empty:
   - Display the root's data.
   - Recursively call `preorder(root->left)`.
   - Recursively call `preorder(root->right)`.

Function: `postorder(root)`

1. If the tree is not empty:
   - Recursively call `postorder(root->left)`.
   - Recursively call `postorder(root->right)`.
   - Display the root's data.

Function: `delete_node(root, x)`

1. If the tree is empty, return `NULL`.
2. Traverse the tree to find the node with the value `x`:
   - If `x > root->data`, recursively call `delete_node(root->right, x)`.
   - If `x < root->data`, recursively call `delete_node(root->left, x)`.
3. If the node is found:
   - If the node has no children, free the node and return `NULL`.
   - If the node has one child, replace it with its child and free the node.
   - If the node has two children:
     - Find the in-order successor using `successor(node)`.
     - Replace the node's data with the successor's data.

  - Recursively delete the successor node from the right subtree.

4. Return the modified tree.

Function: `successor(node)`

1. Start from the right child of the node.
2. Traverse left until the leftmost node is reached.
3. Return the leftmost node as the in-order successor.

Function: `search(key, root)`

1. If the tree is empty or the key matches the root's data, return the root.
2. Traverse the tree:
   - If `key > root->data`, search the right subtree.
   - If `key < root->data`, search the left subtree.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
struct node {
    struct node *left;
    int data;
    struct node *right;
};
void add_node();
void traverse(struct node *root);
void inorder(struct node *root);
void preorder(struct node *root);
void postorder(struct node *root);
struct node *search(int key, struct node *root);
struct node *delete_node(struct node *root, int x);
struct node *successor(struct node *node);
void free_tree(struct node *root);
```

```
struct node *root = NULL;

int main() {

    int is_running = 1, ch;

    while (is_running) {

        printf("\n1. Add Node\n2. Traverse Tree\n3. Delete Node\n4.
        Search\n5. Exit\nChoose an option: ");

        scanf("%d", &ch);


        switch (ch) {

            case 0:

                is_running = 0;

                break;

            case 1:

                add_node();

                break;

            case 2:

                if (root == NULL) {

                    printf("Tree is empty.\n");

                } else {

                    traverse(root);

                }

                break;

            case 3: {

                if (root == NULL) {

                    printf("Tree is empty.\n");

                } else {

                    int element;

                    printf("Enter element to delete: ");

                    scanf("%d", &element);

                    root = delete_node(root, element);
```

```
            }
            break;
        }
        case 4: {
            if (root == NULL) {
                printf("Tree is empty.\n");
            } else {
                int key;
                printf("Enter element to search: ");
                scanf("%d", &key);
                struct node *result = search(key, root);
                if (result != NULL) {
                    printf("Element %d found in the tree.\n", key);
                } else {
                    printf("Element %d not found in the tree.\n");
                }
            }
            break;
        }
        case 5:
            exit(0);

        default:
            printf("Invalid choice. Please try again.\n");
    }
  }
  return 0;
}
void add_node() {
  struct node *newnode;
```

```
    int num;
    newnode = malloc(sizeof(struct node));
    if (newnode == NULL) {
        printf("Memory allocation failed.\n");
        return;
    }
    printf("Enter the data to be added: ");
    scanf("%d", &num);

    newnode->data = num;
    newnode->left = NULL;
    newnode->right = NULL;

    if (root == NULL) {
        root = newnode;
    } else {
        struct node *current = root;
        while (1) {
            if (num == current->data) {
                printf("Element already exists.\n");
                free(newnode);
                return;
            } else if (num > current->data) {
                if (current->right == NULL) {
                    current->right = newnode;
                    break;
                }
                current = current->right;
            } else {
                if (current->left == NULL) {
```

```
                    current->left = newnode;
                    break;
                }
                current = current->left;
            }
        }
        printf("Node inserted.\n");
    }
}
void traverse(struct node *root) {
    int choice;
    printf("\nSelect Traversal Type:\n");
    printf("1. Inorder\n2. Preorder\n3. Postorder\nChoose an option: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            printf("Inorder Traversal: ");
            inorder(root);
            break;
        case 2:
            printf("Preorder Traversal: ");
            preorder(root);
            break;
        case 3:
            printf("Postorder Traversal: ");
            postorder(root);
            break;
        default:
            printf("Invalid traversal type.\n");
```

```
    }
    printf("\n");
}
void inorder(struct node *root) {
    if (root == NULL) return;
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}
void preorder(struct node *root) {
    if (root == NULL) return;
    printf("%d ", root->data);
    preorder(root->left);
    preorder(root->right);
}
void postorder(struct node *root) {
    if (root == NULL) return;
    postorder(root->left);
    postorder(root->right);
    printf("%d ", root->data);
}
struct node *delete_node(struct node *root, int x) {
    if (root == NULL) return root;

    if (x > root->data) {
        root->right = delete_node(root->right, x);
    } else if (x < root->data) {
        root->left = delete_node(root->left, x);
    } else {
        if (root->left == NULL) {
```

```
            struct node *temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct node *temp = root->left;
            free(root);
            return temp;
        } else {
            struct node *succ = successor(root);
            root->data = succ->data;
            root->right = delete_node(root->right, succ->data);
        }
    }
    return root;
}
struct node *successor(struct node *node) {
    node = node->right;
    while (node->left != NULL) {
        node = node->left;
    }
    return node;
}
struct node *search(int key, struct node *root) {
    if (root == NULL || root->data == key) {
        return root;
    }
    if (key > root->data) {
        return search(key, root->right);
    } else {
        return search(key, root->left);
```

```
    }
}
```

Output:

```
1. Add Node
2. Traverse Tree
3. Delete Node
4. Search
5. Exit
Choose an option: 1
Enter the data to be added: 23

1. Add Node
2. Traverse Tree
3. Delete Node
4. Search
5. Exit
Choose an option: 1
Enter the data to be added: 11
Node inserted.

1. Add Node
2. Traverse Tree
3. Delete Node
4. Search
5. Exit
Choose an option: 1
Enter the data to be added: 30
Node inserted.

1. Add Node
2. Traverse Tree
3. Delete Node
4. Search
5. Exit
Choose an option: 1
Enter the data to be added: 35
Node inserted.

1. Add Node
2. Traverse Tree
3. Delete Node
4. Search
5. Exit
Choose an option: 2

Select Traversal Type:
1. Inorder
2. Preorder
3. Postorder
Choose an option: 1
Inorder Traversal: 11 23 30 35

1. Add Node
2. Traverse Tree
3. Delete Node
4. Search
5. Exit
Choose an option: 4
Enter element to search: 30
Element 30 found in the tree.

1. Add Node
2. Traverse Tree
3. Delete Node
4. Search
5. Exit
Choose an option: 3
Enter element to delete: 30

1. Add Node
2. Traverse Tree
3. Delete Node
4. Search
5. Exit
Choose an option: 2

Select Traversal Type:
1. Inorder
2. Preorder
3. Postorder
Choose an option: 1
Inorder Traversal: 11 23 35
```

Result: The program has executed successfully and required output is obtained.

Date: 29/10/24

Program 10: **Depth First Search**

Aim: Write a program to perform depth first search on graph.

Algorithm:

Step 1: Start.

Step 2: Declare the necessary variables:

- `n` for the number of nodes.
- `A[n][n]` for the adjacency matrix (graph representation).
- `visited[n]` (initialized to 0) to track visited nodes.

Step 3: Prompt the user to input the number of nodes (`n`) and read the value.

Step 4: Initialize the adjacency matrix `A[n][n]` to 0.

Step 5: Prompt the user to input edges between nodes until the user enters `-1`:

- For each edge, read the pair of vertices `v1` and `v2`.
- Check if `v1` and `v2` are valid (0 ≤ `v1`, `v2` < `n`).
  - If valid, update the adjacency matrix:

    `A[v1][v2] = 1;`

    `A[v2][v1] = 1;`
  - If invalid, print an error message and prompt the user again.

Step 6: Prompt the user to input the starting node (`source`) for DFS and read the value.

Step 7: Validate the starting node:

- If the starting node is invalid (not in the range 0 to `n-1`), print an error message and exit.

Step 8: Print the message "DFS -> " to indicate the start of traversal.

Step 9: Call the `DFS` function with the following parameters:

- number of nodes, adjacency matrix, visited array, starting node

Step 10: In the `DFS` function:

- Mark the current node (`source`) as visited by setting `visited[source] = 1`.
- Print the current node.
- For each node `i` (0 to `n-1`), check if:
  - `graph[source][i] == 1` (an edge exists) and `visited[i] == 0` (node `i` is not visited).
- If both conditions are true, recursively call `DFS` for node `i`.

Step 11: After completing DFS, print a newline to end the traversal output.

Step 12: End.

Source Code:

```
#include <stdio.h>

void DFS(int n, int graph[n][n], int visited[], int source){
    printf("%d ", source);
    visited[source] = 1;

    for (int i = 0; i < n; i++)
    {
        if (graph[source][i] == 1 && !visited[i])
        {
            DFS(n, graph, visited, i);
        }
    }
}
int main()
{
    int n;
    printf("Enter the number of nodes: ");
    scanf("%d", &n);

    int visited[n];
    for (int i = 0; i < n; i++)
    {
        visited[i] = 0;
    }
    int graph[n][n];
```

```
    for (int i = 0; i < n; i++){
        for (int j = 0; j < n; j++) {
            graph[i][j] = 0;
        }
    }

    int v1, v2;
    printf("Enter edges (enter -1 to stop):\n");
    while (1) {
        printf("Enter the first endpoint of the edge: ");
        scanf("%d", &v1);
        if (v1 == -1)
            break;
        printf("Enter the second endpoint of the edge: ");
        scanf("%d", &v2);

        if (v1 >= n || v2 >= n || a < 0 || b < 0) {
            printf("Invalid edge. Please enter vertices between 0
           and %d.\n", n - 1);
        } else {
            graph[v1][v2] = 1;
            graph[v2][v1] = 1;
        }
    }
    int source;
    printf("Enter the starting node: ");
    scanf("%d", &source);
    if (source < 0 || source >= n)
    {
        printf("Invalid starting node!\n");
```

```
            return 1;
        }
        printf("DFS -> ");
        DFS(n, graph, visited, source);
        printf("\n");

        return 0;
    }
```

Output:

```
Enter the number of edges: 5
Enter edges
Edge 1
Enter first vertex: 0
Enter second vertex: 1
Edge 2
Enter first vertex: 0
Enter second vertex: 2
Edge 3
Enter first vertex: 2
Enter second vertex: 3
Edge 4
Enter first vertex: 2
Enter second vertex: 4
Edge 5
Enter first vertex: 1
Enter second vertex: 2
Enter the starting node: 1
DFS -> 1 0 2 3 4
```

Result: The program has executed successfully and required output is obtained.

Date: 05/11/24

Program 11: **Breadth First Search**

Aim: Write a program to perform breadth first search algorithm on graph.

Algorithm:

Step 1: Input the number of vertices.

- Check if the number of vertices is valid (greater than 0 and less than or equal to the defined maximum).

Step 2: Create the graph:

- Initialize an adjacency matrix (`graph`) with zeros.
- Input edges until the user enters `-1`.
- For each edge:
  - Input two vertices representing the endpoints of the edge.
  - Validate that the vertices are within range.
  - If valid, set `graph[a][b] = 1` and `graph[b][a] = 1` (for an undirected graph).

Step 3: Display the adjacency matrix:

- Print the adjacency matrix row by row.

Step 4: Perform BFS:

- Initialize a queue (`queue`) and a `visited` array of size `MAX`.
- Mark all vertices as unvisited in the `visited` array.
- Input the starting vertex for BFS.
- Validate the starting vertex.
- Add the starting vertex to the queue and mark it as visited.
- While the queue is not empty:
  - Dequeue a vertex (`pop`) and print it.
  - For each neighbor of the dequeued vertex:
    - If the neighbor is connected and unvisited:
      - Enqueue the neighbor.
      - Mark the neighbor as visited.

Step 5: End program.

Source Code:

```
#include <stdio.h>
#define MAX 100
void displayGraph();
void createGraph();
void bfs();
int graph[MAX][MAX] = {0}, vertices;
void main()
{
    printf("How many vertices are there? : ");
    scanf("%d", &vertices);
    if (vertices <= 0 || vertices > MAX) {
        printf("Invalid number of vertices.\n");
        return;
    }
    createGraph();
    displayGraph();
    bfs();
}
void bfs()
{
    int queue[MAX], visited[MAX], front = 0, rear = 0, pop, start;
    for (int i = 0; i < vertices; i++) {
        visited[i] = 0;
    }
    printf("Enter the vertex to start BFS from: ");
    scanf("%d", &start);

    if (start < 0 || start >= vertices) {
        printf("Invalid start vertex!\n");
        return;
    }
    queue[rear] = start;
    visited[start] = 1;
    printf("BFS Traversal: ");
    while (front <= rear) {
```

```
        pop = queue[front];
        printf("%d ", pop);
        front++;
        for (int i = 0; i < vertices; i++) {
            if (graph[pop][i] == 1 && !visited[i]) {
                rear++;
                queue[rear] = i;
                visited[i] = 1;
            }
        }
    }
    printf("\n");
}
void createGraph()
{
    int a, b;
    printf("Enter edges (enter -1 to stop):\n");
    while (1) {
        printf("Enter the first endpoint of the edge: ");
        scanf("%d", &a);
        if (a == -1)
            break;
        printf("Enter the second endpoint of the edge: ");
        scanf("%d", &b);
        if (a >= vertices || b >= vertices || a < 0 || b < 0) {
            printf("Invalid edge. Please enter vertices between 0
             and %d.\n", vertices - 1);
        } else {
            graph[a][b] = 1;
            graph[b][a] = 1;
        }
    }
}
void displayGraph()
{
    printf("\nAdjacency Matrix:\n");
    for (int i = 0; i < vertices; i++) {
```

```
            for (int j = 0; j < vertices; j++) {
                printf("%d ", graph[i][j]);
            }
            printf("\n");
        }
    }
```

Output:

```
How many vertices are there? : 5
Enter edges (enter -1 to stop):
Enter the first endpoint of the edge: 0
Enter the second endpoint of the edge: 2
Enter the first endpoint of the edge: 0
Enter the second endpoint of the edge: 1
Enter the first endpoint of the edge: 2
Enter the second endpoint of the edge: 3
Enter the first endpoint of the edge: 1
Enter the second endpoint of the edge: 4
Enter the first endpoint of the edge: 3
Enter the second endpoint of the edge: 4
Enter the first endpoint of the edge: -1

Adjacency Matrix:
0 1 1 0 0
1 0 0 0 1
1 0 0 1 0
0 0 1 0 1
0 1 0 1 0
Enter the vertex to start BFS from: 0
BFS Traversal: 0 1 2 4 3
```

Result: The program has executed successfully and required output is obtained.

Date: 05/11/24

Program 12: **Kruskal's Algorithm**

Aim: Write a program to perform depth first search on graph.

Algorithm:

Step 1: Input Number of Vertices

- Ask the user to input the number of vertices in the graph (`vertices`).

Step 2: Graph Creation

- Continuously prompt the user to input edges:

1. Input the first and second endpoints of the edge (`a` and `b`).
2. If the endpoints are valid (within range) and the edge does not already exist:
   - Input the weight of the edge.
   - Update the adjacency matrix `graph[a][b]` and `graph[b][a]` with the weight.
3. Stop taking input when the user enters `-1`.

Step 3: Display the Graph

- Print the adjacency matrix representation of the graph.

Step 4: Kruskal's Algorithm

\- Create a `parent` array where `parent[i] = i` for all vertices (each vertex is its own set initially).

- Initialize `edge_count = 0` to track the edges added to the MST.
- While `edge_count < vertices - 1`:

1. Find the edge with the smallest weight (`min`) in the adjacency matrix that has not yet been processed.
2. Remove the edge by setting its weight to `INT_MAX`.
3. Check if the endpoints of the edge belong to different sets:
   - Use the find function to determine the roots of the sets containing the endpoints.
   - If they are in different sets, merge the sets using the union function, include the edge in the MST, and increment `edge_count`.
   - If they are in the same set, discard the edge to avoid cycles.

Step 5: Union-Find Implementation

- Find Operation:

- Recursively find the root of a set:

`find(x)`: Return `x` if it is its own parent, otherwise recursively call `find(parent[x])`.

- Union Operation:

- Merge two sets by updating the parent of one set's root to the other:

`uni(x, y)`: Find the roots of `x` and `y`. If they are different, set `parent[root_y] = root_x`.

Step 6: Display the Minimum Spanning Tree (MST)

Source Code:

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

int graph[100][100] = {0}, vertices, result[100][100] = {0};

void createGraph()
{
    int a, b, i = 0;
    while (1)
    {
        int weight;
        printf("Enter the first end point of the edge (Enter -1 to exit): ");
        scanf("%d", &a);
        if (a == -1)
            break;
        printf("Enter the second end point of the edge: ");
        scanf("%d", &b);
        if (a >= vertices || b >= vertices || a < 0 || b < 0)
        {
            printf("\ninvalid choice\n");
        }
        else
        {
            if (graph[a][b] > 0) {
                printf("\nThe edge already exists!!\n");
```

```
            }
            else {
                printf("Enter the weight of the edge: ");
                scanf("%d", &weight);
                graph[a][b] = weight;
                graph[b][a] = weight;
            }
        }
    }
}

void displayGraph(int arr[100][100], int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
}

int find(int x,int parent[]){
    if (parent[x] == x)
        return x;
    else
        return find(parent[x], parent);
}

int uni(int x,int y,int parent[]) {
    int u = find(x, parent);
    int v = find(y, parent);
    if(u == v){
        return 0;
    }
    else{
        parent[v] = u;
        return 1;
```

```
    }
}

void kruskal() {
    int edge_count = 0;
    int min, weight, u, v;
    int parent[vertices];
    for(int i = 0; i <vertices; i++){
        parent[i] = i;
    }
    while (edge_count < vertices - 1) {
        min = INT_MAX;
        for (int i = 0; i < vertices; i++) {
            for (int j = i; j < vertices; j++) {
                if (graph[i][j] < min && graph[i][j] != 0) {
                    min = graph[i][j];
                    u = i;
                    v = j;
                }
            }
        }
        if (min == INT_MAX)
            break;
        else {
            graph[u][v] = INT_MAX;
            graph[v][u] = INT_MAX;
            if (uni(u, v, parent)) {
                result[u][v] = min;
                result[v][u] = min;
            }
        }
    }
}

int main() {
    printf("How many vertices are there? :");
```

```
    scanf("%d", &vertices);
    createGraph();
    printf("\nThe graph is: \n");
    displayGraph(graph, vertices);
    kruskal();
    printf("\nThe minimum spanning tree is: \n");
    displayGraph(result, vertices);
}
```

Output:

```
How many vertices are there? :4
Enter the first end point of the edge (Enter -1 to exit): 0
Enter the second end point of the edge: 1
Enter the weight of the edge: 5
Enter the first end point of the edge (Enter -1 to exit): 0
Enter the second end point of the edge: 2
Enter the weight of the edge: 8
Enter the first end point of the edge (Enter -1 to exit): 1
Enter the second end point of the edge: 2
Enter the weight of the edge: 10
Enter the first end point of the edge (Enter -1 to exit): 1
Enter the second end point of the edge: 3
Enter the weight of the edge: 15
Enter the first end point of the edge (Enter -1 to exit): 2
```

```
Enter the second end point of the edge: 3
Enter the weight of the edge: 20
Enter the first end point of the edge (Enter -1 to exit): -1

The graph is:
0 5 8 0
5 0 10 15
8 10 0 20
0 15 20 0

The minimum spanning tree is:
0 5 8 0
5 0 0 15
8 0 0 0
0 15 0 0
```

Result: The program has executed successfully and required output is obtained.

Date: 12/11/24

Program 13: **Prim's Algorithm**

Aim: Write a program to implement prim's algorithm to find minimum spanning tree of graph.

Algorithm:

Step 1: Input:

- Read the number of vertices `n`.
- Initialize an adjacency matrix `adj[n][n]` with edge weights. Replace `0` (indicating no edge) with `INT_MAX` to signify infinite weight.

Step 2: Initialization:

- Create an array `vist[n]` to track visited vertices and initialize all elements to `0`.
- Set initial `min = INT_MAX`, `cost = 0`, and edge counter `e = 1`.

Step 3. Find Initial Minimum Edge:

- Iterate through the adjacency matrix to find the minimum weight edge `(u, v)` among all edges. Update `min`, `u`, and `v` accordingly.
- Mark vertices `u` and `v` as visited (`vist[u] = 1` and `vist[v] = 1`).
- Add the weight of this edge to `cost` and print the edge.

Step 4: Build the MST:

- While the edge count `e` is less than `n - 1`:
  - Set `min = INT_MAX`.
  - For each visited vertex `i`:
    - Check all unvisited vertices `j` connected to `i`.
    - If `adj[i][j] < min` and `vist[j] == 0`, update `min`, `u`, and `v` to represent this edge.
  - If no valid edge is found (`min == INT_MAX`), print that the graph is disconnected and terminate the program.
  - Otherwise:
    - Mark vertex `v` as visited.
    - Add the edge weight to `cost`.
    - Print the edge.
    - Increment the edge counter `e`.

Step 5. Output:

- After the loop, print the total cost of the MST.

Source Code:

```
#include <stdio.h>
#include <limits.h>
int main() {
    int n;
    printf("Enter the Number of Vertices: ");
    scanf("%d", &n);
    int adj[n][n], vist[n], min = INT_MAX, u, v, cost = 0;

    printf("Enter the Cost Adjacency Matrix (Enter 0 for no edge):\n");
    for (int i = 0; i < n; i++) {
        vist[i] = 0;
        for (int j = 0; j < n; j++) {
            printf("Weight[%d][%d]: ", i, j);
            scanf("%d", &adj[i][j]);
            if (adj[i][j] == 0)
                adj[i][j] = INT_MAX;
        }
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (adj[i][j] < min) {
                min = adj[i][j];
                u = i;
                v = j;
            }
        }
    }
    cost += min;
    vist[u] = 1;
    vist[v] = 1;
    printf("SPANNING TREE: EDGES ARE\n");
    printf("{%d, %d} = %d\n", u, v, min);
    int e = 1;

    while (e < n - 1) {
```

```
        min = INT_MAX;
        for (int i = 0; i < n; i++) {
            if (vist[i] == 1) {
                for (int j = 0; j < n; j++) {
                    if (adj[i][j] < min && vist[j] == 0) {
                        min = adj[i][j];
                        u = i;
                        v = j;
                    }
                }
            }
        }
        if (min == INT_MAX) {
            printf("Graph is disconnected. No MST possible.\n");
            return 0;
        }
        cost += min;
        vist[v] = 1;
        printf("{%d, %d} = %d\n", u, v, min);
        e++;
    }
    printf("Total Cost = %d\n", cost);
    return 0;
}
```

Output:

```
Enter the Number of Vertices: 4
Enter the Cost Adjacency Matrix (Enter 0 for no edge):
Weight[0][0]: 0
Weight[0][1]: 5
Weight[0][2]: 8
Weight[0][3]: 0
Weight[1][0]: 5
Weight[1][1]: 0
Weight[1][2]: 10
Weight[1][3]: 15
Weight[2][0]: 8
Weight[2][1]: 10
```

```
Weight[2][2]: 0
Weight[2][3]: 20
Weight[3][0]: 0
Weight[3][1]: 15
Weight[3][2]: 20
Weight[3][3]: 0
SPANNING TREE: EDGES ARE
{0, 1} = 5
{0, 2} = 8
{1, 3} = 15
Total Cost = 28
```

Result: The program has executed successfully and required output is obtained.

Date: 12/11/24

Program 14: **Topological sort**

Aim: Write a program to find topological sorting for a directed graph.

Algorithm:

Step 1. Input the Number of Vertices:

- Prompt the user to enter the number of vertices `n`.
- Validate the input; if `n` is less than or equal to 0 or greater than `MAX`, terminate the program with an error message.

Step 2. Initialize Data Structures:

- Create a 2D adjacency matrix `adj[MAX][MAX]` initialized to 0 to store the edges of the graph.
- Create a `visited` array of size `MAX`, initialized to 0, to track visited vertices.

Step 3. Input Edges:

- Repeatedly prompt the user to input pairs of vertices `(v1, v2)` representing directed edges.
- Stop when `v1` is `-1`.
- Validate each edge to ensure:
  - Both `v1` and `v2` are within the range `[0, n-1]`.
  - If valid, set `adj[v1][v2] = 1` to indicate the directed edge.

Step 4. Topological Sorting:

- Repeat the following for `count` from `0` to `n-1`:
  - Set a flag `found` to `0`.
  - Iterate through all vertices `i` from `0` to `n-1`:
    - If vertex `i` is not visited:
      - Check if there are no incoming edges to `i` (i.e., `adj[j][i]` is `0` for all `j`).
      - If no incoming edges are found:
        - Mark vertex `i` as visited.
        - Remove all outgoing edges from `i` (set `adj[i][k] = 0` for all `k`).
        - Print vertex `i`.
        - Set `found = 1` and break the loop.
  - If no vertex was found with zero incoming edges (`found == 0`), a cycle exists:
    - Print a cycle detection message.

- Terminate the program.

Step 5. Output the Result:

- If all vertices are processed successfully, the program outputs the topological order of the vertices.

Source Code:

```
#include <stdio.h>
#define MAX 100

int main() {
    int n, adj[MAX][MAX] = {0}, visited[MAX] = {0}, v1, v2;

    printf("Enter the Number of Vertices: ");
    scanf("%d", &n);

    if (n <= 0 || n > MAX) {
        printf("Invalid number of vertices.\n");
        return 1;
    }
    printf("Enter edges (enter -1 to stop):\n");
    while (1) {
        printf("Enter the first endpoint of the edge: ");
        scanf("%d", &v1);
        if (v1 == -1)
            break;

        printf("Enter the second endpoint of the edge: ");
        scanf("%d", &v2);
        if (v1 >= n || v2 >= n || v1 < 0 || v2 < 0) {
            printf("Invalid edge. Please enter vertices between 0
             and %d.\n", n - 1);
```

```
        } else {
            adj[v1][v2] = 1;
        }
    }
    printf("Topological Sorting is: ");
    for (int count = 0; count < n; count++) {
        int found = 0;
        for (int i = 0; i < n; i++) {
            if (!visited[i]) {
                int flag = 0;

                for (int j = 0; j < n; j++) {
                    if (adj[j][i] == 1) {
                        flag = 1;
                        break;
                    }
                }
                if (!flag) {
                    visited[i] = 1;
                    for (int k = 0; k < n; k++) {
                        adj[i][k] = 0;
                    }
                    printf("%d ", i);
                    found = 1;
                    break;
                }
            }
        }
        if (!found) {
            printf("\nCycle detected. Topological sorting not possible.\n");
```

```
            return 1;
        }
    }
    return 0;
}
```

Output:

```
Enter the Number of Vertices: 4
Enter edges (enter -1 to stop):
Enter the first endpoint of the edge: 0
Enter the second endpoint of the edge: 1
Enter the first endpoint of the edge: 0
Enter the second endpoint of the edge: 2
Enter the first endpoint of the edge: 1
Enter the second endpoint of the edge: 2
Enter the first endpoint of the edge: 1
Enter the second endpoint of the edge: 3
Enter the first endpoint of the edge: 2
Enter the second endpoint of the edge: 3
Enter the first endpoint of the edge: -1
Topological Sorting is: 0 1 2 3
```

Result: The program has executed successfully and required output is obtained.

Date: 19/11/24

Program 15: **Dijkstra's Algorithm**

Aim: Write a program to find shortest path between nodes in a weighted graph.

Algorithm:

Step 1. Input the Graph:

- Read the number of nodes (`n`) and edges (`edges`) in the graph.
- Initialize a 2D adjacency matrix `graph[n][n]` to store edge weights. Set all values to `0` (no edge).

Step 2. Build the Graph:

- For each edge:
  - Read the vertices `v1` and `v2`, and the edge weight `weight`.
  - Update `graph[v1][v2]` and `graph[v2][v1]` with the `weight`.

Step 3. Initialize Variables:

- Create an array `dist[n]` to store the shortest distance from the source to each node. Initialize all values to `INT_MAX` (infinity).
- Create an array `processed[n]` to mark nodes that have been processed. Initialize all values to `0` (not processed).
- Set the distance of the source node `src` to `0`.

Step 4. Find Shortest Path:

- Repeat `n-1` times:
  1. Select the node `u` with the smallest distance (`dist[u]`) that has not been processed.
  2. Mark node `u` as processed (`processed[u] = 1`).
  3. For each neighboring node `v` of `u`:
     - If `v` is not processed and the edge from `u` to `v` exists:
       - Update `dist[v]` to the smaller value between `dist[v]` and `dist[u] + graph[u][v]`.

5. Output Results:

- For each node `i`:
  - If `dist[i] == INT_MAX`, print "Not Reachable".
  - Otherwise, print the shortest distance from the source to node `i`.

Source Code:

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

#define MAX 100

int minDistance(int dist[], int processed[], int n)
{
    int min = INT_MAX, min_index;

    for (int v = 0; v < n; v++)
        if (processed[v] == 0 && dist[v] <= min)
            min = dist[v], min_index = v;

    return min_index;
}

void dijkstra(int graph[MAX][MAX], int src, int n) {
    int dist[n];

    int processed[n];

    for (int i = 0; i < n; i++)
        dist[i] = INT_MAX, processed[i] = 0;

    dist[src] = 0;

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

        int u = minDistance(dist, processed, n);
```

```
        processed[u] = 1;

        for (int v = 0; v < n; v++){
             if (!processed[v] && graph[u][v] && dist[u] != INT_MAX &&
             dist[u] + graph[u][v] < dist[v]){
                  dist[v] = dist[u] + graph[u][v];
            }
        }
    }

    printf("Vertex \t\t Distance from Source\n");
    for (int i = 0; i < n; i++) {
        if (dist[i] == INT_MAX)
            printf("%d \t\t\t\t Not Reachable\n", i);
        else
            printf("%d \t\t\t\t %d\n", i, dist[i]);
    }
}

void main(){
    int nodes, edges, graph[MAX][MAX] = {0};

    printf("Number of nodes: ");
    scanf("%d", &nodes);
    printf("Number of edges: ");
    scanf("%d", &edges);

    int v1, v2, weight, source;

    for (int i = 0; i < edges; i++) {
        printf("Enter v1 v2 and weight: ");
        scanf("%d %d %d", &v1, &v2, &weight);
```

```
            graph[v1][v2] = weight;
            graph[v2][v1] = weight;
        }

        printf("Enter source: ");
        scanf("%d", &source);

        dijkstra(graph, source, nodes);
    }
```

Output:

```
Number of nodes: 7
Number of edges: 8
Enter v1 v2 and weight: 0 2 6
Enter v1 v2 and weight: 0 1 2
Enter v1 v2 and weight: 1 3 5
Enter v1 v2 and weight: 2 3 8
Enter v1 v2 and weight: 3 4 10
Enter v1 v2 and weight: 3 5 15
Enter v1 v2 and weight: 4 6 2
Enter v1 v2 and weight: 5 6 6
Enter source: 0
Vertex          Distance from Source
0                               0
1                               2
2                               6
3                               7
4                               17
5                               22
6                               19
```

Result: The program has executed successfully and required output is obtained.