Experiment No: 1 **Date: 16/10/2024**

SINGLY LINKED STACK

AIM

To implement a stack data structure using a singly linked list and perform basic operations such as push, pop, search, and display.

ALGORITHM

Step 1 : Start.

Step 2 : Display the menu and get user input.

1 : Push

2 : Pop

3 : Search

4 : Display

5 : Exit

Step 3 : Accept the user's choice and validate it.

Step 3.1 : If the choice is valid, proceed to the corresponding operation.

Step 3.2 : If the choice is invalid, display "Invalid choice!" and prompt the user to re-enter.

Step 4 : Perform the push operation if selected.

Step 4.1 : Accept an item to be added to the stack.

Step 4.2 : Dynamically allocate memory for a new node.

Step 4.3 : Assign the entered value to the new node's data field.

Step 4.4 : Link the new node to the current top of the stack.

Step 4.5 : Update the top pointer to point to the new node.

Step 4.6 : Display the stack contents.

Step 5 : Perform the pop operation if selected.

Step 5.1 : Check if the stack is empty by verifying if top is NULL.

Step 5.2 : If empty, display "Stack empty!" and return to the main menu.

Step 5.3 : Otherwise, Store the current top node in a temporary pointer.

Step 5.4 : Update the top pointer to the next node in the stack.

Step 5.5 : Display the data of the removed node.

Step 5.6 : Free the memory allocated to the removed node.

Step 6 : Perform the search operation if selected.

Step 6.1 : Accept the item to search for.

Step 6.2 : Initialize a temporary pointer to traverse the stack from top.

Step 6.3 : Traverse the stack until the end (NULL).

Step 6.4 : Compare the data in the current node with the item.

Step 6.5 : If found, display "Item found".

Step 6.6 : If the traversal completes without finding the item, display "Item not found!".

Step 7 : Perform the display operation if selected.

Step 7.1 : Check if the stack is empty by verifying if top is NULL.

Step 7.2 : If empty, display "Stack empty!".

Step 7.3 : Otherwise, Traverse the stack from top.

Step 7.4 : Display the data of each node sequentially.

Step 8 : Perform the exit operation if selected.

Step 8.1 : Terminate the program.

Step 9 : Handle invalid choices.

Step 9.1 : If the user enters an invalid choice, display "Invalid choice!" and return to the main menu.

Step 10 : Repeat steps 2 to 8 until the user chooses to exit.

Step 11 : Stop.

SOURCE CODE

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

void push()
{
    int value;
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    printf("Enter the value to be inserted:");
    scanf("%d", &value);
    temp->data = value;
    temp->next = top;
    top = temp;
    printf("%d  is inserted", value);
}

void pop()
{
    int item;
    struct node *temp = top;
    if (top == NULL)
    {
        printf("Stack is empty");
    }
```

```
    else
    {
        item = top->data;
        temp = top;
        top = top->next;
        free(temp);
        printf("element popped from the stack");
    }
}

void display()
{
    struct node *temp = top;
    if (top == NULL)
    {
        printf("Stack is empty");
    }
    else
    {
        printf("The Elements in Stack are:\n");
        while (temp != NULL)
        {
            printf("%d->", temp->data);
            temp = temp->next;
        }
    }
}

void search()
{
    struct node *temp = top;
    int pos = 1, found = 0, elem;
```

```
    if (top == NULL)
    {
        printf("Stack is empty");
    }
    printf("Enter the Element to search:");
    scanf("%d", &elem);
    while (temp != NULL)
    {
        if (temp->data == elem)
        {
            printf("%d found at position %d", elem, pos);
            found = 1;
            break;
        }
        temp = temp->next;
        pos++;
    }
    if (found == 0)
    {
        printf("%d is not found in stack", elem);
    }
}

int main()
{
    int choice;
    printf("\nStack operations using Linked List\n");

    do
    {
        printf("\n......................\n1.Push\n2.pop\n3.display\n4.Search\n5.Exit\nEnter your
choice:");
```

```
        scanf("%d", &choice);
        switch (choice)
        {
            case 1:
                push();
                break;
            case 2:
                pop();
                break;
            case 3:
                display();
                break;
            case 4:
                search();
                break;
            case 5:
                printf("Exiting from Stack");
                break;
            default:
                printf("Invalid choice");
                break;
        }
    } while (choice != 5);
    return 0;
}
```

RESULT

The program to implement Stack implementation using linked list is successfully executed and the output is verified.

Experiment No: 2 **Date: 23/10/2024**

SINGLY LINKED LIST

AIM

To implement and demonstrate basic operations on a singly linked list, including insertion, deletion and traversal.

ALGORITHM

Step 1 : Start.

Step 2 : Define the node structure.

Step 2.1 : Declare an integer data field.

Step 2.2 : Declare a pointer to the next node.

Step 3 : Initialize the head pointer to NULL.

Step 4 : Define the createnode function.

Step 4.1 : Accept an item as input.

Step 4.2 : Dynamically allocate memory for a new node.

Step 4.3 : Assign the item to the new node's data field.

Step 4.4 : Set the next pointer of the new node to NULL.

Step 4.5 : Return the new node.

Step 5 : Define the insertAtFront function.

Step 5.1 : Call createnode to create a new node.

Step 5.2 : Set the next pointer of the new node to the current head.

Step 5.3 : Update the head pointer to the new node.

Step 6 : Define the insertAtBack function.

Step 6.1 : Call createnode to create a new node.

Step 6.2 : If the head is NULL, set the head to the new node.

Step 6.3 : Otherwise, traverse the list until the last node.

Step 6.4 : Set the next pointer of the last node to the new node.

Step 7 : Define the insertAtPosition function.

Step 7.1 : Check if the position is valid (position >= 1).

Step 7.2 : If position is 1, call insertAtFront.

Step 7.3 : Otherwise, traverse the list to the position - 1.

Step 7.4 : Create a new node.

Step 7.5 : Set the next pointer of the new node to the current node at the position.

Step 7.6 : Set the next pointer of the previous node to the new node.

Step 8 : Define the deleteFromFront function.

Step 8.1 : Check if the list is empty (head is NULL).

Step 8.2 : If the list is not empty, store the head node in a temporary pointer.

Step 8.3 : Update the head pointer to the next node.

Step 8.4 : Free the memory of the temporary pointer.

Step 9 : Define the deleteFromPosition function.

Step 9.1 : Check if the list is empty (head is NULL).

Step 9.2 : Check if the position is valid (position >= 1).

Step 9.3 : If position is 1, call deleteFromFront.

Step 9.4 : Otherwise, traverse the list to the position - 1.

Step 9.5 : Set the next pointer of the previous node to the next node of the current node.

Step 9.6 : Free the memory of the current node.

Step 10 : Define the deleteFromBack function.

Step 10.1 : Check if the list is empty (head is NULL).

Step 10.2 : If the list contains only one node, free the head and set it to NULL.

Step 10.3 : Otherwise, traverse the list to the second-last node.

Step 10.4 : Set the next pointer of the second-last node to NULL.

Step 10.5 : Free the memory of the last node.

Step 11 : Define the display function.

Step 11.1 : Check if the list is empty (head is NULL).

Step 11.2 : If the list is not empty, traverse the list and print each node's data.

Step 11.3 : After the traversal, print "NULL" to indicate the end of the list.

Step 12 : In the main function, implement a loop to display the menu and get user input.

Step 13 : Implement a switch-case structure to handle user choices.

Step 13.1 : For each case, prompt the user for necessary inputs and call the corresponding function.

Step 13.2 : For invalid choices, display "Invalid choice!" and prompt the user again.

Step 14 : Repeat steps 12 to 13 until the user chooses to exit.

Step 15 : Stop.

SOURCE CODE

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

struct node{
    int data;
    struct node* next;
};
struct node* head = NULL;
struct node* createnode(int data){
    struct node* newnode = (struct node*)malloc(sizeof(struct node));
    newnode->data = data;
    newnode->next = NULL;
    return newnode;
}

void insertatbeginning(int data){
    struct node* newnode = createnode(data);
    newnode->next = head;
    head = newnode;
}

void insertatend(int data){
    struct node* newnode = createnode(data);
```

```
    if (head == NULL){
        head = newnode;
        return;
    }
    struct node* temp = head;
    while (temp->next != NULL)
        temp = temp->next;
    temp->next = newnode;
}

void insertatposition(int data, int position){
    if (position < 1){
        printf("Position should be >= 1.\n");
        return;
    }
    if (position == 1){
        insertatbeginning(data);
        return;
    }
    struct node* newnode = createnode(data);
    struct node* temp = head;
    for (int i = 1; i < position - 1 && temp != NULL; i++){
        temp = temp->next;
    }
    if (temp == NULL){
        printf("Position is greater than the length of the list.\n");
        free(newnode);
    } else{
        newnode->next = temp->next;
        temp->next = newnode;
    }
}
```

```
void deleteatbeginning(){
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct node* temp = head;
    head = head->next;
    free(temp);
}

void deleteatend(){
    if (head == NULL){
        printf("List is empty.\n");
        return;
    }
    struct node* temp = head;
    if (temp->next == NULL){
        free(temp);
        head = NULL;
        return;
    }
    struct node* prev = NULL;
    while (temp->next != NULL){
        prev = temp;
        temp = temp->next;
    }
    prev->next = NULL;
    free(temp);
}

void deleteatposition(int position){
    if (head == NULL){
```

```
        printf("List is empty.\n");
        return;
    }
    if (position < 1){
        printf("Position should be >= 1.\n");
        return;
    }
    if (position == 1){
        deleteatbeginning();
        return;
    }
    struct node* temp = head;
    struct node* prev = NULL;
    for (int i = 1; i < position && temp != NULL; i++){
        prev = temp;
        temp = temp->next;
    }
    if (temp == NULL){
        printf("Position is greater than the length of the list.\n");
    } else {
        prev->next = temp->next;
        free(temp);
    }
}

void searchelement(int data) {
    struct node* temp = head;
    int position = 1;
    while (temp != NULL) {
        if (temp->data == data) {
            printf("Element %d found at position %d\n", data, position);
            return;
```

```
        }
        temp = temp->next;
        position++;
    }
    printf("Element %d not found in the list.\n", data);
}

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

int main() {
    int choice, data, position;

    do {
       printf("\n....Singly LInkedList OPerations....\n");
        printf("\nSelect an Operation:\n");
        printf("1. Insert at Beginning\n");
        printf("2. Insert at End\n");
        printf("3. Insert at Position\n");
        printf("4. Delete at Beginning\n");
        printf("5. Delete at End\n");
        printf("6. Delete at Position\n");
```

```
printf("7. Search an Element\n");
printf("8. Display list\n");
printf("9. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
   case 1:
      printf("Enter data to insert at beginning: ");
      scanf("%d", &data);
      insertatbeginning(data);
      break;
   case 2:
      printf("Enter data to insert at end: ");
      scanf("%d", &data);
      insertatend(data);
      break;
   case 3:
      printf("Enter data to insert: ");
      scanf("%d", &data);
      printf("Enter position to insert: ");
      scanf("%d", &position);
      insertatposition(data, position);
      break;
   case 4:
      deleteatbeginning();
      break;
   case 5:
      deleteatend();
      break;
```

```
        case 6:
            printf("Enter position to delete: ");
            scanf("%d", &position);
            deleteatposition(position);
            break;

        case 7:
            printf("Enter element to search: ");
            scanf("%d", &data);
            searchelement(data);
            break;

        case 8:
            displayList();
            break;

        case 9:
            printf("Exited...\n");
            break;

        default:
            printf("Invalid choice, try again.\n");
      }
   } while (choice != 9);
   return 0;
}
```

RESULT

The program to implement singly linked list and its operations is successfully executed and the output is verified.

Experiment No: 3 **Date: 30/10/2024**

DOUBLY LINKED LIST

AIM

To implement and demonstrate basic operations on a Doubly Linked list, including insertion (at the beginning, end, and specific position), deletion (from the beginning, end, and specific position), and traversal.

ALGORITHM

Step 1 : Start.

Step 2 : Define the node structure.

Step 2.1 : Declare a pointer prev for the previous node.

Step 2.2 : Declare an integer data for the node's data.

Step 2.3 : Declare a pointer next for the next node.

Step 3 : Initialize the head pointer to NULL.

Step 4 : Define the createnode function.

Step 4.1 : Accept an item as input.

Step 4.2 : Dynamically allocate memory for a new node.

Step 4.3 : Set the prev pointer of the new node to NULL.

Step 4.4 : Set the data of the new node to the input item.

Step 4.5 : Set the next pointer of the new node to NULL.

Step 4.6 : Return the new node.

Step 5 : Define the insertAtFront function.

Step 5.1 : Call createnode to create a new node.

Step 5.2 : Set the next pointer of the new node to the current head.

Step 5.3 : Update the head pointer to the new node.

Step 5.4 : Print "Item Inserted!".

Step 6 : Define the insertAtPosition function.

Step 6.1 : Check if the position is valid (position >= 1).

Step 6.2 : If position is 1, call insertAtFront.

Step 6.3 : Otherwise, traverse the list to position - 1.

Step 6.4 : Create a new node.

Step 6.5 : Set the next pointer of the new node to the current node at the position.

Step 6.6 : Set the prev pointer of the new node to the previous node.

Step 6.7 : Set the next pointer of the previous node to the new node.

Step 6.8 : Set the prev pointer of the next node to the new node.

Step 6.9 : Print "Item Inserted at position X!".

Step 7 : Define the insertAtBack function.

Step 7.1 : Call createnode to create a new node.

Step 7.2 : If the head is NULL, set the head to the new node.

Step 7.3 : Otherwise, traverse the list to the last node.

Step 7.4 : Set the next pointer of the last node to the new node.

Step 7.5 : Set the prev pointer of the new node to the last node.

Step 7.6 : Set the next pointer of the new node to NULL.

Step 7.7 : Print "Item Inserted!".

Step 8 : Define the deleteFromFront function.

Step 8.1 : Check if the list is empty (head is NULL).

Step 8.2 : If the list is not empty, store the head node in a temporary pointer.

Step 8.3 : Update the head pointer to the next node.

Step 8.4 : Set the prev pointer of the new head node to NULL.

Step 8.5 : Free the memory of the temporary pointer.

Step 8.6 : Print "Item Deleted!".

Step 9 : Define the deleteFromPosition function.

Step 9.1 : Check if the list is empty (head is NULL).

Step 9.2 : Check if the position is valid (position >= 1).

Step 9.3 : If position is 1, call deleteFromFront.

Step 9.4 : Otherwise, traverse the list to the position - 1.

Step 9.5 : Set the next pointer of the previous node to the next node of the current node.

Step 9.6 : Set the prev pointer of the next node to the previous node.

Step 9.7 : Free the memory of the current node.

Step 9.8 : Print "Item Deleted!".

Step 10 : Define the deleteFromBack function.

Step 10.1 : Check if the list is empty (head is NULL).

Step 10.2 : If the list contains only one node, free the head and set it to NULL.

Step 10.3 : Otherwise, traverse the list to the second-last node.

Step 10.4 : Set the next pointer of the second-last node to NULL.

Step 10.5 : Free the memory of the last node.

Step 10.6 : Print "Item Deleted!".

Step 11 : Define the display function.

Step 11.1 : Check if the list is empty (head is NULL).

Step 11.2 : If the list is not empty, traverse the list and print each node's data.

Step 11.3 : After the traversal, print "NULL" to indicate the end of the list.

Step 12 : In the main function, implement a loop to display the menu and get user input.

Step 13 : Implement a switch-case structure to handle user choices.

Step 13.1 : If the user chooses option 1 (Insert at front), prompt the user to enter the item and call insertAtFront.

Step 13.2 : If the user chooses option 2 (Insert at position), prompt the user to enter the item and position, and call insertAtPosition.

Step 13.3 : If the user chooses option 3 (Insert at back), prompt the user to enter the item and call insertAtBack.

Step 13.4 : If the user chooses option 4 (Delete from front), call deleteFromFront().

Step 13.5 : If the user chooses option 5 (Delete from position), prompt the user to enter the position and call deleteFromPosition.

Step 13.6 : If the user chooses option 6 (Delete from back), call deleteFromBack().

Step 13.7 : If the user chooses option 7 (Display), call display().

Step 13.8 : If the user chooses option 8 (Exit), display "Exiting..." and exit the program.

Step 14 : Repeat steps 12 and 13 until the user chooses to exit.

Step 15 : Stop.

SOURCE CODE

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
	struct node * prev;
	int data;
	struct node * next;
};
struct node * head=NULL;

struct node * createnode(int item)
{
	struct node * newnode = (struct node*)malloc(sizeof(struct node));
	newnode->prev = NULL;
	newnode->data = item;
	newnode->next = NULL;
	return newnode;
}

void insertAtFront(int item)
{
	struct node * newnode = createnode(item);
	newnode->next = head;
	head = newnode;
	printf("%d Inserted!\n",item);
}

void insertAtPosition(int item,int position)
{
	if(position<1)
```

```
	{
		printf("Position cannot be <1.\n");
		return;
	}
	if(position == 1)
	{
		insertAtFront(item);
		return;
	}
	struct node * newnode = createnode(item);
	struct node * temp = head;
	for (int i = 1; i < position - 1 && temp != NULL; i++)
	{
		temp = temp->next;
	}
	if (temp == NULL)
	{
		printf("Position Not available.\n");
		free(newnode);
	}
	else
	{
		newnode->next = temp->next;
		newnode->prev = temp;
		temp->next = newnode;
		temp->next->prev = newnode;
		printf("Item Inserted at %d position!\n",position);
	}
}

void insertAtBack(int item)
{
```

```
        struct node * newnode = createnode(item);
        if(head == NULL)
        {
                head = newnode;
        }
        struct node* temp = head;
        while(temp->next != NULL)
                temp = temp->next;
                temp->next = newnode;
                newnode->prev = temp;
                newnode->next=NULL;
                printf("%d Inserted!\n",item);
}

void deleteFromFront()
{
    if(head == NULL)
    {
        printf("List is empty!\n");
        return;
    }
    struct node* temp = head;
    head = temp->next;
    if (head != NULL)
    {
        head->prev = NULL;
    }
    free(temp);
    printf("First Node Deleted!\n");
}
```

```
void deleteFromPosition(int position)
{
        if (head == NULL)
        {
                printf("List is empty.\n");
                return;
        }
        if (position < 1)
        {
                printf("Position cannot be <1.\n");
                return;
        }
        if (position == 1)
        {
                deleteFromFront();
                return;
        }
        struct node* temp = head;
        struct node* loc = NULL;
        for (int i = 1; i < position && temp != NULL; i++)
        {
                loc = temp;
                temp = temp->next;
        }
        if (temp == NULL)
        {
                printf("Position Not available.\n");
        }
        else
        {
                loc->next = temp->next;
                temp->next->prev = loc;
```

```
            free(temp);
            printf("Item deleted");
        }
}

void deleteFromBack()
{
        if(head == NULL)
        {
            printf("List is empty.\n");
            return;
        }
        struct node* temp = head;
        struct node* loc = NULL;
        if(temp->next == NULL)
        {
            free(temp);
            head = NULL;
            return;
        }
        while(temp->next != NULL)
        {
            loc=temp;
            temp = temp->next;
        }
        loc->next = NULL;
        free(temp);
        printf("Last Node Deleted!\n");
}
```

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

int main()
{
    int choice,item,position;
    do{
            printf("\n*************DOUBLY LINKEDLIST
            OPERATION****************\n\n");
        printf("1.Insert at Beginning\n2.Insert at End\n3.Insert at Position\n4.Delete from
        Beginning\n5.Delete from End\n");
        printf("6.Delete from Position\n7.Display\n8.Exit\nEnter your choice :");
        scanf("%d",&choice);
        switch(choice)
        {
            case(1):
                printf("Enter item to Insert be  at Beginning of the List :");
                scanf("%d",&item);
```

```
        insertAtFront(item);
        break;
    case(2):
        printf("Enter item to be Insert at End :");
        scanf("%d",&item);
        insertAtBack(item);
        break;
    case(3):
        printf("Enter the position :");
        scanf("%d",&position);
        printf("Enter item to be inserted :");
        scanf("%d",&item);
        insertAtPosition(item,position);
        break;
    case(4):
        deleteFromFront();
        break;
    case(5):
        deleteFromBack();
        break;
    case(6):
        printf("Enter the position :");
        scanf("%d",&position);
        deleteFromPosition(position);
        break;
    case(7):
        display();
        break;
```

```
            case(8):
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice!Try again.\n");
                break;
        }
    } while(choice!=8);
    return 0;
}
```

RESULT

The program to implement doubly linked list and its operations is successfully executed and the output is verified.

Experiment No: 4 **Date: 13/11/2024**

BINARY SEARCH TREE

AIM

To implement various operations on a Binary Search Tree (BST), including insertion, search, traversal (inorder, preorder, postorder), deletion, and display.

ALGORITHM

Step 1 : Start.

Step 2 : Define the node structure.

Step 2.1 : Declare a pointer left for the left child node.

Step 2.2 : Declare a pointer right for the right child node.

Step 2.3 : Declare an integer data for the node's data.

Step 3 : Initialize the root pointer to NULL.

Step 4 : Define the newnode function.

Step 4.1 : Accept a value as input.

Step 4.2 : Dynamically allocate memory for a new node.

Step 4.3 : Set the data of the new node to the input value.

Step 4.4 : Set the left and right pointers of the new node to NULL.

Step 4.5 : Return the new node.

Step 5 : Define the insert function.

Step 5.1 : If the root is NULL, create a new node and return it.

Step 5.2 : If the value is equal to the root’s data, print "Same data can't be stored!" and return the root.

Step 5.3 : If the value is greater than the root’s data, recursively insert the value in the right subtree.

Step 5.4 : If the value is smaller than the root’s data, recursively insert the value in the left subtree.

Step 5.5 : Return the root after insertion.

Step 6 : Define the inorderTraversal function.

Step 6.1 : If the root is NULL, return.

Step 6.2 : Recursively traverse the left subtree.

Step 6.3 : Print the data of the current node.

Step 6.4 : Recursively traverse the right subtree.

Step 7 : Define the preorderTraversal function.

Step 7.1 : If the root is NULL, return.

Step 7.2 : Print the data of the current node.

Step 7.3 : Recursively traverse the left subtree.

Step 7.4 : Recursively traverse the right subtree.

Step 8 : Define the postorderTraversal function.

Step 8.1 : If the root is NULL, return.

Step 8.2 : Recursively traverse the left subtree.

Step 8.3 : Recursively traverse the right subtree.

Step 8.4 : Print the data of the current node.

Step 9 : Define the searchNode function.

Step 9.1 : If the root is NULL, print "Item does not Found!" and return NULL.

Step 9.2 : If the root's data matches the value, print "Item Found in Tree!" and return the node.

Step 9.3 : If the root's data is less than the value, recursively search in the right subtree.

Step 9.4 : If the root's data is greater than the value, recursively search in the left subtree.

Step 10 : Define the minValueNode function.

Step 10.1 : Initialize a temporary pointer temp to the root.

Step 10.2 : Traverse left while the left child exists.

Step 10.3 : Return the leftmost node.

Step 11 : Define the deleteNode function.

Step 11.1 : If the root is NULL, print "tree is Empty!" and return NULL.

Step 11.2 : If the value is smaller than the root's data, recursively delete from the left subtree.

Step 11.3 : If the value is greater than the root's data, recursively delete from the right subtree.

Step 11.4 : If the value matches the root’s data, check the following cases:

Step 11.5 : If the node has no left child, return the right child after freeing the current node.

Step 11.6 : If the node has no right child, return the left child after freeing the current node.

Step 11.7 : If the node has both children, find the minimum value node in the right subtree, replace the root’s data with it, and recursively delete the minimum value node.

Step 12 : In the main function, display the menu and prompt the user for input.

Step 13 : Implement a switch-case structure to handle user choices.

Step 13.1 : If the user chooses option 1 (Insert Node), prompt the user to enter the value and call insert.

Step 13.2 : If the user chooses option 2 (Search Node), prompt the user to enter the value and call searchNode.

Step 13.3 : If the user chooses option 3 (Inorder Traversal), call inorderTraversal.

Step 13.4 : If the user chooses option 4 (Preorder Traversal), call preorderTraversal.

Step 13.5 : If the user chooses option 5 (Postorder Traversal), call postorderTraversal.

Step 13.6 : If the user chooses option 6 (Delete Node), prompt the user to enter the value and call deleteNode.

Step 13.7 : If the user chooses option 7 (Exit), display "Exiting..." and exit the program.

Step 14 : Repeat steps 12 and 13 until the user chooses to exit.

Step 15 : Stop.

SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>
struct node {
    int data;
    struct node* left;
    struct node* right;
};
struct node *root = NULL;

void createNode(int x) {
    struct node *newnode = (struct node*)malloc(sizeof(struct node));
    newnode->data = x;
    newnode->left = NULL;
    newnode->right = NULL;
    if (root == NULL) {
        root = newnode;
        printf("Created root node: %d\n", root->data);
    } else {
        printf("Node created: %d\n", newnode->data);
    }
}

void insert(int item) {
    if (root == NULL) {
        createNode(item);
        return;
    }
    struct node *current = root;
    struct node *parent = NULL;
    while (1) {
```

```
        parent = current;
        if (item < current->data) {
            current = current->left;
            if (current == NULL) {
                parent->left = (struct node*)malloc(sizeof(struct node));
                parent->left->data = item;
                parent->left->left = parent->left->right = NULL;
                printf("Inserted %d to the left of %d\n", item, parent->data);
                return;
            }
        } else {
            current = current->right;
            if (current == NULL) {
                parent->right = (struct node*)malloc(sizeof(struct node));
                parent->right->data = item;
                parent->right->left = parent->right->right = NULL;
                printf("Inserted %d to the right of %d\n", item, parent->data);
                return;
            }
        }
    }
}

struct node* minValueNode(struct node* node) {
    struct node* current = node;
    while (current && current->left != NULL)
        current = current->left;
    return current;
}
struct node* del(struct node* root, int item) {
    if (root == NULL) {
        printf("Node %d not found in the tree.\n", item);
```

```
        return NULL;
    }
    if (item < root->data)
        root->left = del(root->left, item);
    else if (item > root->data)
        root->right = del(root->right, item);
    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;
        }
        struct node* temp = minValueNode(root->right);
        root->data = temp->data;
        root->right = del(root->right, temp->data);
    }
    return root;
}

void search(int item) {
    struct node *current = root;
    while (current != NULL) {
        if (current->data == item) {
            printf("Node %d found\n", item);
            return;
        }
        current = (item < current->data) ? current->left : current->right;
    }
```

```
    printf("Node %d not found in the tree.\n", item);
}

void inorderTraversal(struct node *root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d  ", root->data);
        inorderTraversal(root->right);
    }
}

void preorderTraversal(struct node *root) {
    if (root != NULL) {
        printf("%d  ", root->data);
        preorderTraversal(root->left);
        preorderTraversal(root->right);
    }
}

void postorderTraversal(struct node *root) {
    if (root != NULL) {
        postorderTraversal(root->left);
        postorderTraversal(root->right);
        printf("%d  ", root->data);
    }
}

int main() {
    while (1) {
        int ch;
            printf("\n********* BST OPERATIONS **********\n");
```

```
printf("\n1.Insert a Node\n2.Delete a Node\n3.Search a Node\n4.Inorder
Traversal\n5.Preorder Traversal\n6.Postorder traversal\n7.Exit\n\nEnter your choice: ");

scanf("%d", &ch);

switch (ch) {

        case 1:

            int x1;

            printf("\nEnter the key to be inserted: ");

            scanf("%d", &x1);

            insert(x1);

            break;

        case 2:

            int x2;

            printf("\nEnter the key to be deleted: ");

            scanf("%d", &x2);

            root = del(root, x2);

            break;

        case 3:

            int x3;

            printf("\nEnter the key to be searched: ");

            scanf("%d", &x3);

            search(x3);

            break;

        case 4:

            printf("Inorder Traversal: ");

            inorderTraversal(root);

            printf("\n");

            break;

        case 5:

            printf("Preorder Traversal: ");

            preorderTraversal(root);

            printf("\n");
```

```
                break;
            case 6:
                printf("Postorder Traversal: ");
                postorderTraversal(root);
                printf("\n");
                break;
            case 7:
                printf("\nExiting !\n");
                exit(0);
            default:
                printf("INVALID CHOICE !\n");
        }
    }
    return 0;
}
```

RESULT

The program to implement binary search tree using linked list is successfully executed and the output is verified.

Experiment No: 5 **Date: 13/11/2024**

CIRCULAR QUEUE

AIM

To implement a circular queue using a dynamic array and perform operations like enqueue, dequeue, search, and display.

ALGORITHM

Step 1 : Start.

Step 2 : Define variables:

Step 2.1 : front initialized to -1.

Step 2.2 : rear initialized to -1.

Step 2.3 : Declare a pointer queue for the circular queue.

Step 2.4 : Declare an integer item for the element to be enqueued or dequeued.

Step 2.5 : Declare an integer size for the size of the queue.

Step 2.6 : Declare an integer i for iteration.

Step 3 : Prompt the user to enter the size of the circular queue.

Step 4 : Allocate memory for the queue dynamically.

Step 4.1 : If memory allocation fails, print an error message and exit.

Step 5 : Define the enqueue function.

Step 5.1 : If both front and rear are -1, set front and rear to 0.

Step 5.2 : If the queue is full (when (rear + 1) % size == front), print "Queue is Full".

Step 5.3 : Otherwise, prompt the user to enter the element, increment rear, and insert the element at the new rear position.

Step 6 : Define the dequeue function.

Step 6.1 : If both front and rear are -1 (empty queue), print "Queue Underflow".

Step 6.2 : If front equals rear, print the deleted element, and reset front and rear to -1 (empty queue).

Step 6.3 : Otherwise, print the deleted element and increment front using (front + 1) % size.

Step 7 : Define the display function.

Step 7.1 : If both front and rear are -1 (empty queue), print "Nothing to Display".

Step 7.2 : Otherwise, iterate through the queue starting from front to rear, printing each element.

Step 8 : Define the search function.

Step 8.1 : If both front and rear are -1, print "Queue is Empty".

Step 8.2 : Otherwise, prompt the user to enter the element to search for.

Step 8.3 : Iterate through the queue from front to rear and check if the element is present.

Step 8.4 : If the element is found, print "Element found".

Step 8.5 : If the element is not found, print "Element not found".

Step 9 : In the main function, display the menu and prompt the user for input.

Step 10 : Implement a switch-case structure to handle user choices.

Step 10.1 : If the user chooses option 1 (Enqueue), call enqueue.

Step 10.2 : If the user chooses option 2 (Dequeue), call dequeue.

Step 10.3 : If the user chooses option 3 (Display), call display.

Step 10.4 : If the user chooses option 4 (Search), call search.

Step 10.5 : If the user chooses option 5 (Exit), free the allocated memory for the queue and exit.

Step 11 : Repeat steps 9 and 10 until the user chooses to exit.

Step 12 : Stop

SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>
int *queue;
int size;
int front = -1, rear = -1;

void initializeQueue() {
    queue = (int *)malloc(size * sizeof(int));
}

void enqueue(int element) {
    if (front == (rear + 1) % size) {
        printf("Queue is full\n");
        return;
    }
    if (front == -1 && rear == -1) {
        front = rear = 0;
    } else {
        rear = (rear + 1) % size;
    }
    queue[rear] = element;
}

int dequeue() {
    int element;
    if (front == -1 && rear == -1) {
        printf("Queue is empty\n");
        return -1;
    }
    element = queue[front];
```

```
    if (front == rear) {
        front = rear = -1;
    } else {
        front = (front + 1) % size;
    }
    printf("%d dequeued from the queue\n", element);
    return element;
}

int searchElement(int element) {
    if (front == -1 && rear == -1) {
        printf("Queue is empty\n");
        return -1;
    }
    int current = front;
    int position = 1;
    do {
        if (queue[current] == element) {
            return position;
        }
        current = (current + 1) % size;
        position++;
    } while (current != (rear + 1) % size);
    return -1;
}

void displayQueue() {
    if (front == -1 && rear == -1) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements: ");
```

```
    int current = front;
    do {
        printf("%d ", queue[current]);
        current = (current + 1) % size;
    } while (current != (rear + 1) % size);
    printf("\n");
}

int main() {
    int choice, searchResult, element;
    printf("Enter the size of the Circular Queue: ");
    scanf("%d", &size);
    initializeQueue();

    do {
        printf("\n*Circular Queue Operations*\n");
        printf("\n1. ENQUEUE\n");
        printf("2. DEQUEUE\n");
        printf("3. SEARCH\n");
        printf("4. DISPLAY\n");
        printf("5. EXIT\n");
        printf("Enter Your choice: ");
        scanf("%d", &choice);

        switch (choice) {
                case 1:
                    printf("Enter the element to Enqueue: ");
                    scanf("%d", &element);
                    enqueue(element);
                    break;
```

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

            case 3:
                printf("Enter the element to search: ");
                scanf("%d", &element);
                searchResult = searchElement(element);
                if (searchResult != -1) {
                    printf("%d found at position %d\n", element, searchResult);
                } else {
                    printf("%d not found in the queue\n", element);
                }
                break;

            case 4:
                displayQueue();
                break;

            case 5:
                printf("Exiting the program.\n");
                break;

            default:
                printf("Invalid choice. Please enter a valid option.\n");
                break;
        }
    } while (choice != 5);
    free(queue);
    return 0;
}
```

RESULT

The program to implement circular queue is successfully executed and the output is verified**.**

Experiment No: 6 **Date: 04/12/2024**

SET DATA STRUCTURE

AIM

To implement bit strings for performing set operations like Union, Intersection, and Difference.

ALGORITHM

Step 1 : Start.

Step 2 : Define constants and variables:

Step 2.1 : MAX_SIZE for the maximum set size.

Step 2.2 : Arrays for superSet, setA, setB, bitStringA, and bitStringB.

Step 2.3 : Integers superSetSize, setASize, setBSize for the sizes of sets.

Step 3 : Define function getUniversalSet().

Step 3.1 : Prompt the user for the size of the Universal Set and validate.

Step 3.2 : Ask for the elements of the Universal Set and store them.

Step 4 : Define function getSet().

Step 4.1 : Prompt the user to enter the elements of a set, ensuring all elements are in the Universal Set.

Step 5 : Define function checkSetInUniversal().

Step 5.1 : Check if each element of the set is in the Universal Set. If any element is not found, print an error and return 0.

Step 5.2 : Return 1 if all elements are valid.

Step 6 : Define function generateBitStrings().

Step 6.1 : Initialize bit strings for sets A and B.

Step 6.2 : For each element in set A, set the corresponding bit in bitStringA.

Step 6.3 : For each element in set B, set the corresponding bit in bitStringB.

Step 6.4 : Print the bit strings for sets A and B.

Step 7 : Define function setUnion().

Step 7.1 : Perform the union operation using the bitwise OR (|) operator.

Step 7.2 : Print the result of the union in both bit string and set form.

Step 8 : Define function setIntersection().

Step 8.1 : Perform the intersection operation using the bitwise AND (&)operator.

Step 8.2 : Print the result of the intersection in both bit string and set form.

Step 9 : Define function setDifferenceAminusB().

Step 9.1 : Perform the difference operation A - B using the bitwise AND (&) operator and negation of bitStringB.

Step 9.2 : Print the result of the difference (A - B) in both bit string and set form.

Step 10 : Define function setDifferenceBminusA().

Step 10.1 : Perform the difference operation B - A using the bitwise AND (&) operator and negation of bitStringA.

Step 10.2 : Print the result of the difference (B - A) in both bit string and set form.

Step 11 : Define function printBitString().

Step 11.1 : Print the bit string in a readable format.

Step 12 : Define function printSetFromBitString().

Step 12.1 : Convert the bit string back to a set and print the corresponding elements.

Step 13 : In main(), call getUniversalSet() to get the Universal Set.

Step 14 : Prompt the user to input Set A and Set B, ensuring that their sizes do not exceed the Universal Set size and that all elements are valid.

Step 15 : Call generateBitStrings() to generate the bit strings for Set A and Set B.

Step 16 : Display the menu for set operations (Union, Intersection, Difference).

Step 16.1 : If the user chooses Union, call setUnion().

Step 16.2 : If the user chooses Intersection, call setIntersection().

Step 16.3 : If the user chooses Difference A - B, call setDifferenceAminusB().

Step 16.4 : If the user chooses Difference B - A, call setDifferenceBminusA().

Step 16.5 : If the user chooses to Exit, exit the program.

Step 17 : Repeat Step 16 until the user chooses to Exit.

Step 18 : Stop.

SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 20

int superSet[MAX_SIZE], superSetSize = 0;
int setA[MAX_SIZE], setASize = 0;
int setB[MAX_SIZE], setBSize = 0;
int bitStringA[MAX_SIZE], bitStringB[MAX_SIZE];

void getUniversalSet();
void getSet(int arr[], int *size);
int checkSetInUniversal(int arr[], int size);
void generateBitStrings();
void setUnion();
void setIntersection();
void setDifferenceAminusB();
void setDifferenceBminusA();
void printBitString(int arr[], int size);
void printSetFromBitString(int arr[], int size);

void getUniversalSet() {
        printf("\n***** BIT STRING OPERATIONS *******\n");
  printf("  Enter Universal Set Size : ");
  scanf("%d", &superSetSize);
  if (superSetSize > MAX_SIZE) {
    printf("Error: Size exceeds maximum limit.\n");
    exit(1);
  }
  printf("  Enter %d elements for the Universal Set:\n", superSetSize);
  for (int i = 0; i < superSetSize; i++) {
```

```
        printf("Element %d: ", i + 1);
        scanf("%d", &superSet[i]);
    }
}

void getSet(int arr[], int *size) {
    printf("Enter %d elements (must be in the Universal Set):\n", *size);
    for (int i = 0; i < *size; i++) {
        printf("Element %d: ", i + 1);
        scanf("%d", &arr[i]);
    }
}

int checkSetInUniversal(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        int found = 0;
        for (int j = 0; j < superSetSize; j++) {
            if (arr[i] == superSet[j]) {
                found = 1;
                break;
            }
        }
        if (!found) {
            printf("Error: Element %d is not in the Universal Set. Please enter the set again.\n", 
arr[i]);
            return 0;
        }
    }
    return 1;
}
```

```
void generateBitStrings() {
    for (int i = 0; i < superSetSize; i++) {
        bitStringA[i] = 0;
        bitStringB[i] = 0;
    }
    for (int i = 0; i < setASize; i++) {
        for (int j = 0; j < superSetSize; j++) {
            if (setA[i] == superSet[j]) {
                bitStringA[j] = 1;
                break;
            }
        }
    }
    for (int i = 0; i < setBSize; i++) {
        for (int j = 0; j < superSetSize; j++) {
            if (setB[i] == superSet[j]) {
                bitStringB[j] = 1;
                break;
            }
        }
    }
    printf("\nSet A Bit String: ");
    printBitString(bitStringA, superSetSize);
    printf("Set B Bit String: ");
    printBitString(bitStringB, superSetSize);
}

void setUnion() {
    int bitStringUnion[MAX_SIZE];
    for (int i = 0; i < superSetSize; i++) {
        bitStringUnion[i] = bitStringA[i] | bitStringB[i];
    }
```

```
    printf("Union: ");
    printBitString(bitStringUnion, superSetSize);
    printf("Union Values: ");
    printSetFromBitString(bitStringUnion, superSetSize);
}

void setIntersection() {
    int bitStringIntersection[MAX_SIZE];
    for (int i = 0; i < superSetSize; i++) {
        bitStringIntersection[i] = bitStringA[i] & bitStringB[i];
    }
    printf("Intersection: ");
    printBitString(bitStringIntersection, superSetSize);
    printf("Intersection Values: ");
    printSetFromBitString(bitStringIntersection, superSetSize);
}

void setDifferenceAminusB() {
    int bitStringDifferenceAminusB[MAX_SIZE];
    for (int i = 0; i < superSetSize; i++) {
        bitStringDifferenceAminusB[i] = bitStringA[i] & (1 - bitStringB[i]);
    }
    printf("Difference (A - B): ");
    printBitString(bitStringDifferenceAminusB, superSetSize);
    printf("Difference Result (A - B): ");
    printSetFromBitString(bitStringDifferenceAminusB, superSetSize);
}

void setDifferenceBminusA() {
    int bitStringDifferenceBminusA[MAX_SIZE];
    for (int i = 0; i < superSetSize; i++) {
        bitStringDifferenceBminusA[i] = bitStringB[i] & (1 - bitStringA[i]);
```

```
    }
    printf("Difference (B - A): ");
    printBitString(bitStringDifferenceBminusA, superSetSize);
    printf("Difference Result (B - A): ");
    printSetFromBitString(bitStringDifferenceBminusA, superSetSize);
}

void printBitString(int arr[], int size) {
    printf("{");
    for (int i = 0; i < size; i++) {
        printf("%d", arr[i]);
        if (i < size - 1) {
            printf(", ");
        }
    }
    printf("}\n");
}

void printSetFromBitString(int arr[], int size) {
    int first = 1;
    printf("{");
    for (int i = 0; i < size; i++) {
        if (arr[i] == 1) {
            if (!first) {
                printf(", ");
            }
            printf("%d", superSet[i]);
            first = 0;
        }
    }
    printf("}\n");
}
```

```
int main() {
    int choice;
    getUniversalSet();
    do {
        printf("Enter Set A Size [max size %d]: ", superSetSize);
        scanf("%d", &setASize);
        if (setASize > superSetSize) {
            printf("Error: Set A size cannot exceed Universal Set size.\n");
        }
    } while (setASize > superSetSize);

    do {
        getSet(setA, &setASize);
    } while (checkSetInUniversal(setA, setASize) == 0);
    do {
        printf("Enter Set B Size [max size %d]: ", superSetSize);
        scanf("%d", &setBSize);
        if (setBSize > superSetSize) {
            printf("Error: Set B size cannot exceed Universal Set size.\n");
        }
    } while (setBSize > superSetSize);
    do {
        getSet(setB, &setBSize);
    } while (checkSetInUniversal(setB, setBSize) == 0);
    generateBitStrings();
    do {
        printf("\n\nChoose a Set  Operation:\n");
        printf("1. Union of A and B\n");
        printf("2. Intersection of A and B\n");
        printf("3. Difference (A - B)\n");
        printf("4. Difference (B - A)\n");
```

```
        printf("5. Exit\n\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
                case 1:
                    setUnion();
                    break;
                case 2:
                    setIntersection();
                    break;
                case 3:
                    setDifferenceAminusB();
                    break;
                case 4:
                    setDifferenceBminusA();
                    break;
                case 5:
                    printf("Exiting...\n");
                    break;
                default:
                    printf("Invalid choice. Please try again.\n");
            }
        } while (choice != 5);
  return 0;
}
```

RESULT

The program to implement set implementation using bit string is successfully executed and the output is verified.

Experiment No: 7 **Date: 07/12/2024**

DISJOINT SETS

AIM

To implement a disjoint set data structure using linked lists for performing set operations like union, find, and display of disjoint sets.

ALGORITHM

Step 1 : Start.

Step 2 : Define structures and global variables:

Step 2.1 : Define struct node with fields rep (representative), next (pointer to next node), and data (element value).

Step 2.2 : Define global arrays heads and tails to hold the head and tail of each set, and a counter countRoot to track the number of sets.

Step 3 : Define function makeSet(int x).

Step 3.1 : Allocate memory for a new node with element x.

Step 3.2 : Set the node's representative to itself and its next pointer to NULL.

Step 3.3 : Add the new node to the heads and tails arrays.

Step 4 : Define function find(int a).

Step 4.1 : Traverse the sets and find the representative of element a.

Step 4.2 : Return the representative node if found.

Step 4.3 : If the element is not found, return NULL.

Step 5 : Define function unionSets(int a, int b).

Step 5.1 : Find the representatives of elements a and b.

Step 5.2 : If either representative is NULL, print an error message.

Step 5.3 : If the representatives are different, merge the two sets by attaching the second set to the first.

Step 5.4 : Update the representatives of all nodes in the second set to the first set's representative.

Step 6 : Define function search(int x).

Step 6.1 : Check if element x exists in any set.

Step 6.2 : Return 1 if the element is found, otherwise return 0.

Step 7 : Define function displayRepresentatives().

Step 7.1 : Print the representative of each set.

Step 8 : Define function displaySets().

Step 8.1 : Print all sets in their entirety, displaying all elements in each set.

Step 9 : In main(), prompt the user for the size of the set and validate the input.

Step 10 : Prompt the user to enter the elements of the set, ensuring they are unique.

Step 11 : Display the menu for set operations (display representatives, union, find set, display all sets, exit).

Step 11.1 : If the user chooses to display representatives, call displayRepresentatives().

Step 11.2 : If the user chooses union, prompt for two elements and call unionSets().

Step 11.3 : If the user chooses to find a set, prompt for an element and call find().

Step 11.4 : If the user chooses to display all sets, call displaySets().

Step 11.5 : If the user chooses to exit, terminate the program.

Step 12 : Repeat Step 11 until the user exits.

Step 13 : Stop.

SOURCE CODE

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

struct node {
   struct node *rep;
   struct node *next;
   int data;
}*heads[50],*tails[50];
static int countRoot = 0;
```

```
void makeSet(int x) {
    struct node *new=(struct node *)malloc(sizeof(struct node));
    new->rep=new;
    new->next=NULL;
    new->data=x;
    heads[countRoot]=new;
    tails[countRoot]=new;
    countRoot++;
}

struct node* find(int a) {
    int i;
    struct node *tmp;
    for (i=0;i<countRoot;i++) {
        tmp=heads[i];
        while (tmp!=NULL) {
            if (tmp->data==a)
                return tmp->rep;
            tmp=tmp->next;
        }
    }
    return NULL;
}

void unionSets(int a, int b) {
    int i, j, pos, flag = 0;
    struct node *tail2;
    struct node *rep1=find(a);
    struct node *rep2=find(b);
    if (rep1==NULL || rep2==NULL) {
        printf("\nElement not present\n");
        return;
```

```
    }
    if (rep1!=rep2) {
        for (j=0;j<countRoot;j++) {
            if (heads[j]==rep2) {
                pos=j;
                flag=1;
                countRoot-=1;
                tail2=tails[j];
                for (i=pos;i<countRoot;i++) {
                    heads[i]=heads[i+1];
                    tails[i]=tails[i+1];
                }
                break;
            }
        }
        for (j=0;j<countRoot;j++) {
            if (heads[j]==rep1) {
                tails[j]->next=rep2;
                tails[j]=tail2;
                break;
            }
        }
        while (rep2!=NULL) {
            rep2->rep=rep1;
            rep2=rep2->next;
        }
    }
}

int search(int x) {
    int i;
    struct node *tmp;
```

```
    for (i=0;i<countRoot;i++) {
        tmp=heads[i];
        while (tmp!=NULL) {
            if (tmp->data==x)
                return 1;
            tmp=tmp->next;
        }
    }
    return 0;
}

void displayRepresentatives() {
    printf("\nRepresentative Elements: ");
    for (int i=0;i<countRoot;i++) {
        printf("%d ", heads[i]->data);
    }
    printf("\n");
}

void displaySets() {
    int i, j;
    struct node *temp;
    printf("\nDisjoint Sets:\n");
    for (i=0;i<countRoot;i++) {
        temp=heads[i];
        printf("{ ");
        int first=1;
        while (temp!=NULL) {
            if (!first) printf(", ");
            printf("%d", temp->data);
            first=0;
            temp=temp->next;
```

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

int main() {
    int choice, x, y, setSize;
    printf("\n*** UNION AND FIND USING DISJOINT SET ***\n");
    printf("\nEnter the size of the set : ");
    scanf("%d", &setSize);
    while (setSize<=0 || setSize>30) {
        printf("Please enter a size between 1 and 30: ");
        scanf("%d", &setSize);
    }
    printf("\nEnter %d unique elements for the set:\n", setSize);
    for (int i = 0; i < setSize; ) {
        printf("Enter element %d: ", i + 1);
        scanf("%d", &x);
                if (search(x)) {
            printf("Element %d already exists in the set. Please enter a unique element.\n", x);
        } else {
            makeSet(x);
            i++;
        }
    }
    do {
        printf("\n\t**OPERATIONS**\n");
        printf("\n1. Display Sets");
        printf("\n2. Display Representatives");
        printf("\n3. Set Union");
        printf("\n4. Find element");
        printf("\n5. Exit");
```

```
printf("\nEnter your choice: ");
scanf("%d", &choice);
switch(choice) {
        case 1:
          displaySets();
          break;
        case 2:
          displayRepresentatives();
          break;
        case 3:
          printf("\nEnter first element: ");
          scanf("%d", &x);
          printf("Enter second element: ");
          scanf("%d", &y);
          unionSets(x, y);
          break;
        case 4:
          printf("\nEnter the element to find: ");
          scanf("%d", &x);
          struct node *rep = find(x);
          if (rep == NULL) {
            printf("\nElement not present in the DS\n");
          } else {
            printf("\nThe representative of %d is %d\n", x, rep->data);
          }
          break;
        case 5:
          exit(0);
```

```
            default:
                printf("\nInvalid choice! Please try again.\n");
                break;
        }
    } while (1);
    return 0;
}
```

RESULT

The program to implement disjoint set is successfully executed and the output is verified.

Experiment No: 8 **Date: 18/12/2024**

GRAPH TRAVERSAL TECHNIQUES (DFS AND BFS) AND TOPOLOGICAL SORTING

AIM

To implement graph operations such as Breadth-First Search (BFS), Depth-First Search (DFS), and Topological Sort using an adjacency list representation.

ALGORITHM

Step 1 : Start.

Step 2 : Define global variables:

Step 2.1 : Define adj_matrix[MAX][MAX] for the adjacency matrix.

Step 2.2 : Define visited[MAX] for keeping track of visited vertices.

Step 2.3 : Define vertex_count to store the number of vertices in the graph.

Step 3 : Define function add_vertex().

Step 3.1 : Check if vertex_count is less than MAX.

Step 3.2 : If yes, increment vertex_count and print that a vertex has been added.

Step 3.3 : If no, print a message indicating the maximum number of vertices is reached.

Step 4 : Define function add_edge(int start, int end).

Step 4.1 : Check if start and end are valid vertex indices.

Step 4.2 : If valid, set adj_matrix[start][end] = 1 to add an edge and print the edge.

Step 4.3 : If invalid, print an error message.

Step 5 : Define function print_adj_matrix().

Step 5.1 : Print the adjacency matrix showing the relationships between vertices.

Step 6 : Define function bfs(int start).

Step 6.1 : Initialize visited array to false and the queue.

Step 6.2 : Mark start as visited and enqueue it.

Step 6.3 : While the queue is not empty, dequeue a vertex and print it.

Step 6.4 : For each adjacent vertex of the dequeued vertex, if it is unvisited, mark it visited and enqueue it.

Step 7 : Define function dfs_helper(int vertex).

Step 7.1 : Mark the vertex as visited and print it.

Step 7.2 : For each adjacent vertex of the current vertex, if unvisited, recursively call dfs_helper.

Step 8 : Define function dfs(int start).

Step 8.1 : Reset the visited array.

Step 8.2 : Call dfs_helper(start) to perform DFS starting from start.

Step 8.3 : For each unvisited vertex, call dfs_helper to ensure all disconnected components are covered.

Step 9 : Define function topological_sort_helper(int vertex, int stack[], int *top).

Step 9.1 : Mark the vertex as visited.

Step 9.2 : For each adjacent vertex of the current vertex, if unvisited, recursively call topological_sort_helper.

Step 9.3 : Push the vertex to the stack once all its adjacent vertices are processed.

Step 10 : Define function topological_sort().

Step 10.1 : Reset the visited array.

Step 10.2 : Call topological_sort_helper for each unvisited vertex.

Step 10.3 : Print the vertices in reverse order from the stack to get the topological sort.

Step 11 : In main(), display the menu and get user input.

Step 11.1 : If the user chooses to add a vertex, call add_vertex().

Step 11.2 : If the user chooses to add an edge, call add_edge(start, end).

Step 11.3 : If the user chooses BFS, call bfs(start).

Step 11.4 : If the user chooses DFS, call dfs(start).

Step 11.5 : If the user chooses topological sort, call topological_sort().

Step 11.6 : If the user chooses to print the adjacency matrix, call print_adj_matrix().

Step 11.7 : If the user chooses to exit, terminate the program.

Step 12 : Repeat Step 11 until the user exits.

Step 13 : Stop.

SOURCE CODE

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 10

int adj_matrix[MAX][MAX];
bool visited[MAX];
int vertex_count = 0;

void add_vertex() {
    if (vertex_count >= MAX) {
        printf("Cannot add more vertices. Maximum reached.\n");
        return;
    }
    vertex_count++;
    printf("Vertex %d added.\n", vertex_count - 1);
}

void add_edge(int start, int end) {
    if (start >= vertex_count || end >= vertex_count) {
        printf("Invalid vertex index!\n");
        return;
    }
    adj_matrix[start][end] = 1;
    printf("Edge added from %d to %d.\n", start, end);
}
```

```
void bfs(int start) {
    bool visited[MAX] = {false};
    int queue[MAX], front = 0, rear = 0;
    visited[start] = true;
    queue[rear++] = start;
    printf("BFS: ");
    while (front < rear) {
        int vertex = queue[front++];
        printf("%d ", vertex);
        for (int i = 0; i < vertex_count; i++) {
            if (adj_matrix[vertex][i] && !visited[i]) {
                visited[i] = true;
                queue[rear++] = i;
            }
        }
    }
    printf("\n");
}

void dfs_helper(int vertex) {
    visited[vertex] = true;
    printf("%d ", vertex);
    for (int i = 0; i < vertex_count; i++) {
        if (adj_matrix[vertex][i] && !visited[i]) {
            dfs_helper(i);
        }
    }
}
```

```
void dfs(int start) {
    for (int i = 0; i < MAX; i++) visited[i] = false;
    printf("DFS: ");
    dfs_helper(start);
    printf("\n");
      for (int i = 0; i < vertex_count; i++) {
        if (!visited[i]) {
            printf("DFS starting from vertex %d: ", i);
            dfs_helper(i);
            printf("\n");
        }
    }
}

void topological_sort_helper(int vertex, int stack[], int *top) {
    visited[vertex] = true;
    for (int i = 0; i < vertex_count; i++) {
        if (adj_matrix[vertex][i] && !visited[i]) {
            topological_sort_helper(i, stack, top);
        }
    }
    stack[(*top)++] = vertex;
}

void topological_sort() {
    for (int i = 0; i < MAX; i++) visited[i] = false;
    int stack[MAX], top = 0;
    for (int i = 0; i < vertex_count; i++) {
        if (!visited[i]) {
            topological_sort_helper(i, stack, &top);
        }
    }
```

```
    printf("Topological Sort: ");
    while (top > 0) {
        printf("%d ", stack[--top]);
    }
    printf("\n");
}


int main() {
    int choice, start, end, vertex;
   printf("\n\tGRAPH TRAVERSAL- BFS & DFS\n");
    do {
        printf("\n1. Add Vertex\n2. Add Edge\n3. BFS\n4. DFS\n5. Topological Sort\n6.
Exit\n\nEnter your choice: ");
        scanf("%d", &choice);


        switch (choice) {
                        case 1:
                            add_vertex();
                            break;
                        case 2:
                            printf("Enter start and end vertex: ");
                            scanf("%d %d", &start, &end);
                            add_edge(start, end);
                            break;
                        case 3:
                            printf("Enter starting vertex for BFS: ");
                            scanf("%d", &vertex);
                            bfs(vertex);
                            break;
```

```
            case 4:
                printf("Enter starting vertex for DFS: ");
                scanf("%d", &vertex);
                dfs(vertex);
                break;
            case 5:
                topological_sort();
                break;
            case 6:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice! Try again.\n");
        }
    } while (choice != 6);
    return 0;
}
```

RESULT

The program to implement the graph traversal of depth first search, breadth first search and topological sort is successfully executed and the output is verified.

Experiment No: 9 **Date: 20/12/2024**

STRONGLY CONNECTED COMPONENTS

AIM

To implement a program to find Strongly Connected Components (SCCs) in a directed graph using Kosaraju's Algorithm.

ALGORITHM

Step 1 : Start.

Step 2 : Define the Graph structure.

Step 2.1 : Define numVertices to store the number of vertices.

Step 2.2 : Define adjMatrix[MAX_VERTICES][MAX_VERTICES] for the adjacency matrix.

Step 2.3 : Define reverseAdjMatrix[MAX_VERTICES][MAX_VERTI CES] for the reversed adjacency matrix.

Step 3 : Define function initGraph(struct Graph* graph, int vertices).

Step 3.1 : Initialize numVertices with the given number of vertices.

Step 3.2 : Initialize both adjacency matrices with 0 (no edges).

Step 4 : Define function addEdge(struct Graph* graph, int src, int dest).

Step 4.1 : Add a directed edge in the original graph by setting adjMatrix[src][dest]=1.

Step 4.2 : Add a reverse edge in the reversed graph by setting reverse- -AdjMatrix[dest][src] = 1.

Step 5 : Define function DFS().

Step 5.1 : Mark the vertex as visited.

Step 5.2 : For each adjacent vertex of the current vertex, if it is unvisited, recursively call DFS.

Step 5.3 : After exploring all adjacent vertices, push the current vertex to the stack.

Step 6 : Define function DFSReverse().

Step 6.1 : Mark the vertex as visited and part of the current SCC.

Step 6.2 : For each adjacent vertex in the reversed graph, if unvisited, re cursively call DFSReverse.

Step 7 : Define function kosarajuSCC(struct Graph* graph).

Step 7.1 : Initialize visited[MAX_VERTICES] to false and stack[MAX_VERTICES].

Step 7.2 : Perform DFS on the original graph and fill the stack with in finishing time order.

Step 7.3 : Reset the visited array.

Step 7.4 : Perform DFS on the reversed graph using the order of vertices in the stack.

Step 7.5 : For each unvisited vertex, perform DFS on the reverse graph to find all nodes in the current SCC.

Step 7.6 : Print each SCC.

Step 8 : Define function displayGraph(struct Graph* graph).

Step 8.1 : Print the adjacency matrix to represent the graph.

Step 9 : In main(), prompt the user to enter the number of vertices and edges.

Step 9.1 : Call initGraph() to initialize the graph.

Step 9.2 : Input edges and call addEdge() to add them to the graph.

Step 10 : Display a menu with options to display the graph, find SCCs, or exit.

Step 10.1 : If the user chooses to display the graph, call displayGraph().

Step 10.2 : If the user chooses to find SCCs, call kosarajuSCC().

Step 10.3 : If the user chooses to exit, terminate the program.

Step 11 : Repeat Step 10 until the user exits.

Step 12 : Stop.

SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX_VERTICES 10

struct Graph {
    int numVertices;
    int adjMatrix[MAX_VERTICES][MAX_VERTICES]; // Adjacency matrix
    int reverseAdjMatrix[MAX_VERTICES][MAX_VERTICES]; // Reversed adjacency
matrix
};
void initGraph(struct Graph* graph, int vertices) {
    graph->numVertices = vertices;
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            graph->adjMatrix[i][j] = 0;
            graph->reverseAdjMatrix[i][j] = 0;
        }
    }
}

void addEdge(struct Graph* graph, int src, int dest) {
    graph->adjMatrix[src][dest] = 1;  // Add edge in the original graph
    graph->reverseAdjMatrix[dest][src] = 1;
}

void DFS(struct Graph* graph, int vertex, bool visited[], int stack[], int* stackIndex, int
adjMatrix[MAX_VERTICES][MAX_VERTICES]) {
    visited[vertex] = true;
```

```
    for (int i = 0; i < graph->numVertices; i++) {
        if (adjMatrix[vertex][i] == 1 && !visited[i]) {
            DFS(graph, i, visited, stack, stackIndex, adjMatrix);
        }
    }
    stack[*stackIndex] = vertex;
    (*stackIndex)++;
}

void DFSReverse(struct Graph* graph, int vertex, bool visited[], int* component, int
reverseAdjMatrix[MAX_VERTICES][MAX_VERTICES]) {
    visited[vertex] = true;
    component[vertex] = 1;
    for (int i = 0; i < graph->numVertices; i++) {
        if (reverseAdjMatrix[vertex][i] == 1 && !visited[i]) {
            DFSReverse(graph, i, visited, component, reverseAdjMatrix);
        }
    }
}

void kosarajuSCC(struct Graph* graph) {
    bool visited[MAX_VERTICES] = { false };
    int stack[MAX_VERTICES];
    int stackIndex = 0;
    for (int i = 0; i < graph->numVertices; i++) {
        if (!visited[i]) {
            DFS(graph, i, visited, stack, &stackIndex, graph->adjMatrix);
        }
    }
    for (int i = 0; i < graph->numVertices; i++) {
        visited[i] = false;  // Reset visited array
    }
```

```
printf("Strongly Connected Components (SCCs):\n");
    while (stackIndex > 0) {
        int vertex = stack[--stackIndex];
        if (!visited[vertex]) {
            int component[MAX_VERTICES] = { 0 }; // To track the SCC
            DFSReverse(graph, vertex, visited, component, graph->reverseAdjMatrix);
            printf("{ ");
            for (int i = 0; i < graph->numVertices; i++) {
                if (component[i]) {
                    printf("%d ", i);
                }
            }
            printf("}\n");
        }
    }
}

void displayGraph(struct Graph* graph) {
    printf("\nGraph Representation (Adjacency Matrix):\n");
    for (int i = 0; i < graph->numVertices; i++) {
        for (int j = 0; j < graph->numVertices; j++) {
            printf("%d ", graph->adjMatrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    struct Graph graph;
    int vertices, edges, src, dest, choice;
```

```
printf("Enter the number of vertices: ");
scanf("%d", &vertices);
initGraph(&graph, vertices);
printf("Enter the number of edges: ");
scanf("%d", &edges);
for (int i = 0; i < edges; i++) {
    printf("Enter edge %d (source destination): ", i + 1);
    scanf("%d %d", &src, &dest);
    if (src >= 0 && src < vertices && dest >= 0 && dest < vertices) {
        addEdge(&graph, src, dest);
    } else {
        printf("Invalid edge! Please enter vertices within the range of 0 to %d.\n",vertices -1);
        i--;
    }
}

printf("STRONGLY CONNECTED GRAPH\n");
do {
    printf("\n**********\n");
    printf("1. Display Graph\n");
    printf("2. Find Strongly Connected Components (SCCs)\n");
    printf("3. Exit\n*********\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
                case 1:
                    displayGraph(&graph);
                    break;
```

```
                case 2:
                    kosarajuSCC(&graph);
                    break;
                case 3:
                    printf("Exiting the program.\n");
                    break;
                default:
                    printf("Invalid choice! Please try again.\n");
        }
    } while (choice != 3);
    return 0;
}
```

RESULT

The program to implement the strongly connected components is successfully executed and the output is verified.

Experiment No: 10 **Date: 20/12/2024**

PRIM'S ALGORITHM

AIM

To implement Prim's Algorithm to find the Minimum Spanning Tree (MST) of a graph, using an adjacency matrix representation.

ALGORITHM

Step 1 : Start.

Step 2 : Ask the user for the number of nodes and edges.

Step 3 : Initialize the cost matrix to represent the graph, setting all values to 999 (representing no edges) and 0 for the diagonal (self-loops).

Step 4 : Input edges and their weights.

  Step 4.1 : For each edge, update the adjacency matrix (cost[][]) for both directions (since the graph is undirected).

Step 5 : Initialize the visited array to track which nodes are included in the MST.

  Step 5.1 : Set the starting node (node 0) as visited.

Step 6 : Begin Prim's Algorithm to find the MST.

  Step 6.1 : Repeat the following until ne (number of edges in the MST) is less than n-1 (the MST has n-1 edges):

  Step 6.2 : For each unvisited node pair (i, j), find the edge with the minimum weight that connects an already visited node to an unvisited node.

  Step 6.3 : If the edge connects an unvisited node, add the edge to the MST, update the visited array, and add the edge's weight to mincost.

  Step 6.4 : Mark the edge as used by setting its weight to 999.

Step 7 : After constructing the MST, print the total weight of the MST.

Step 8 : Stop.

SOURCE CODE

```
#include <stdio.h>
int n, i, j, u, v, a, b;
int cost[10][10], visited[10]= {0}, min, mincost= 0, ne= 1;

void main() {
	printf("\nEnter the number of Vertices: ");
	scanf("%d", &n);
	printf("\nEnter the Adjacency matrix:\n");
	for (i = 0; i < n; i++) {
		for (j = 0; j < n; j++) {
			scanf("%d", &cost[i][j]);
			if (cost[i][j] == 0) {
				cost[i][j] = 999;
			}
		}
	}

	visited[0] = 1;
	printf("\n");
	while (ne < n) {
		for (i = 0, min = 999; i < n; i++) {
			for (j = 0; j < n; j++) {
				if (cost[i][j] < min && visited[i] != 0) {
					min = cost[i][j];
					a = u = i;
					b = v = j;
				}
			}
		}
```

```
            if (visited[u] == 0 || visited[v] == 0) {
            printf("\nEdge %d: (%d %d) cost: %d", ne++, a, b, min);
            mincost += min;
            visited[b] = 1;
        }
        cost[a][b] = cost[b][a] = 999;
    }
    printf("\n\nMinimum cost: %d\n", mincost);
}
```

RESULT

The program to implement the prim’s algorithm using disjoint data structure is successfully executed and the output is verified.

Experiment No: 11 **Date: 30/12/2024**

KRUSKAL’S ALGORITHM

AIM

To implement Kruskal's Algorithm for finding the Minimum Spanning Tree (MST) of a graph, using the Union-Find data structure to detect and avoid cycles.

ALGORITHM

Step 1 : Start.

Step 2 : Ask the user for the number of nodes (n) and edges (m).

Step 3 : Declare an array of edges.

Step 4 : Input the edges with their weights.

    Step 4.1 : For each edge, store the two vertices (u, v) and the weight of the edge in the edges array.

Step 5 : Initialize the Disjoint Set (Union-Find) data structure.

    Step 5.1 : Set each node's parent to itself (parent[i] = i) and initialize the rank (rank[i] = 0).

Step 6 : Sort the edges by their weights using the qsort function and the compareEdges function.

Step 7 : Start processing the sorted edges.

    Step 7.1 : For each edge (u, v): - If the roots of u and v are different then: Add the edge to the MST.

    Step 7.2 : Perform a union operation to combine the sets of u and v.

    Step 7.3 : Update the total weight of the MST.If adding the edge forms a cycle , skip the edge.

Step 8 : Repeat step 7 until n-1 edges are included in the MST.

Step 9 : Print the edges included in the MST and the total weight.

Step 10 : Stop.

SOURCE CODE

```
#include<stdio.h>
#include<stdlib.h>
int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,cost[99][99],parent[99];
int find(int);
int uni(int,int);

int main()
{
	printf("\nEnter the number of vertices: ");
	scanf("%d", &n);
	printf("\nEnter the cost adjacency matrix:\n");
	for(i=0;i<n;i++)
	{
		for(j=0;j<n;j++)
		{
			scanf("%d",&cost[i][j]);
			if(cost[i][j]==0)
				cost[i][j]=999;
		}
	}
	printf("The Minimum Cost Spanning Tree(MST) using Kruskal's Algorithm:\n");
	while(ne<n)
	{
		for(i=1,min=999;i<n;i++)
		{
			for(j=0;j<n;j++)
			{
				if(cost[i][j]<min)
				{
```

```
                                        min=cost[i][j];
                                        a=u=i;
                                        b=v=j;
                                }
                        }
                }
                u=find(u);
                v=find(v);
                if(uni(u,v))
                {
                        printf("Edge %d: (%d,%d)\tcost : %d\n",ne++,a,b,min);
                        mincost+=min;
                }
                cost[a][b]=cost[b][a]=999;
        }
        printf("\nMinimum cost of the MST = %d\n",mincost);
        return 0;
}

int find(int i)
{
        while(parent[i])
                i = parent[i];
        return i;
}

int uni(int i, int j)
{
        if(i!=j)
        {
                parent[j]=i;
                return 1;
```

```
	}
	return 0;
}
```

RESULT

The program to implement the kruskal’s algorithm using disjoint data structure is successfully executed and the output is verified