MASTER OF COMPUTER APPLICATIONS

PRACTICAL RECORD WORK
ON
20MCA135 – DATA STRUTURES LAB

Submitted by

NAME : MUHAMMED ASIF SAJJAD V K

Reg. No. : VDA24MCA-2032

DEPARTMENT OF COMPUTER APPLICATIONS

COLLEGE OF ENGINEERING VATAKARA

(CAPE – GOVT. OF KERALA)

JANUARY 2025

DEPARTMENT OF COMPUTER APPLICATIONS

COLLEGE OF ENGINEERING VATAKARA

CAPE(– GOVT. OF KERALA)

CERTIFICATE

Certified that this is the bonafide record work on the practical course **20MCA135 DATA STRUCTRES LAB** done and prepared Mr./Ms. **MUHAMMED ASIF SAJJAD V K** (**Reg No.VDA24MCA-2032**), 1st Semester MCA(2024-26 Batch) student of Department of Computer Applications at College of Engineering Vatakara, in the partial fulfilment of the award of MCA Degree of APJ Abdul Kalam Technological University (KTU).

Date :

Faculty-in-Charge **Head of the Department**

(Office Seal)

Internal Examiners: **External Examiners:**

INDEX				
Exp. No	**Experiment Details**	**Date**	**Page No**	**Remarks**
1	Array Insertion	19-11-24	4	
2	Array deletion	19-11-24	6	
3	Array Operations	20-11-24	8	
4	Array Merging and Sorting	21-11-24	12	
5	Linked List	22-11-24	14	
6	Stack using Array	25-11-24	18	
7	Queue using Array	26-11-24	22	
8	Stack using Linked List	01-12-24	26	
9	Circular Queue	10-12-24	30	
10	Disjoint set	11-12-24	35	
11	Kruskal’s Algorithm	12-12-24	40	
12	Depth First Search	13-12-24	43	
13	Breadth First Search	15-12-24	45	
14	BST-Inorder	18-12-24	48	
15	Prim’s Algorithm	23-12-24	53	
16	Topological Sort	24-12-24	56	

Experiment No: 1 **Date: 19-11-24**

Aim:
To implement a program in C to insert an element at a specific position in an array.

Algorithm:

Step1:Start

Step 2: Declare an array a[100], variables n, i, data, and pos.

Step 3: Prompt the user to input the number of elements (n) in the array.

Step 4: Read the input and store it in n.

Step 5: Prompt the user to enter n elements of the array.

Step 6: Read and store the elements in the array a.

Step 7: Prompt the user to input the data to be inserted.

Step 8: Read and store the data.

Step 9: Prompt the user to input the pos (position) where the element will be inserted.

Step 10: Read and store the pos.

Step 11: Shift all elements starting from position pos-1 to the right by one position.

Step 12: Insert the data at position pos-1 in the array.

Step 13: Increment the size of the array (n++).

Step 14: Display the updated array.

Step 15: End

Source code:

```
#include<stdio.h>
int main()
{
        int a[100],n,i,data,pos;

        printf("enter the length\n");
        scanf("%d",&n);

        printf("enter the array\n");
         for(i=0;i<n;i++)
        scanf("%d",&a[i]);
       printf("enter data for insertion");
        scanf("%d",&data);
        printf("enter the position");
        scanf("%d",&pos);
```

```
    for(i=n-1;i>=pos-1;i--)
        {
            a[i+1]=a[i];
        }

        a[pos-1]=data;
        n++;
        printf("inserted data is\n");
        for(i=0;i<n;i++)
            printf("%d ",a[i]);

        return 0;
    }
```

Output:

```
enter the length: 4
enter the array: 4
5
2
3
enter data for insertion8
enter the position3
inserted data is
4 5 8 2 3

------------------
(program exited with code: 0)
```

Experiment No: 2 **Date: 19-11-24**

Aim:

To implement a program in C to delete an element from a specific position in an array.

Algorithm:

Step1. Start

Step2. Declare an array a[100] and variables n, i, and pos.

Step3. Prompt the user to enter the number of elements (n) in the array.

Step4. Read the value of n.

Step5. Prompt the user to enter the elements of the array.

Step6. Store the elements in the array a.

Step7. Prompt the user to input the position (pos) of the element to delete.

Step8. Check if pos is greater than n:

- If true, print an error message like "Invalid position."
- Otherwise, proceed to the next step.

Step9. Shift all elements to the left starting from position pos-1 until the second last element of the array (n-1).

Step10. Decrease the size of the array by 1 (n--).

Step11. Print the updated array.

Step12. End

Source code:

```
#include<stdio.h>
int main(){
  int a[100],i,n,pos;
  printf("enter the length of an array\n");
  scanf("%d",&n);
  printf("enter the array\n ");

  for(i=0;i<n;i++)
        scanf("%d",&a[i]);

  printf("enter the position to delete");
  scanf("%d",&pos);

  if(pos > n)
        printf("linavid position");
  else
  {
        for(i=pos-1;i<n-1;i++)
              a[i]=a[i+1];
```

```
}
n--;
for(i=0;i<n;i++)
        printf("%d\n",a[i]);

return 0;
}
```

Output:

```
enter the length of an array
4
enter the array
 11
12
13
14
enter the position to delete3
11
12
14


------------------
(program exited with code: 0)
```

Experiment No: 3 **Date: 20-11-24**

Aim:

To create a menu-driven program in C for performing create, insert, delete, and view operations on an array.

Algorithm:

Step1. Start

Step2. Declare a global array a[100] and variables n and i.

Step3. Define the following functions:

- create():
  1. Prompt the user to enter the number of elements (n).
  2. Read n elements into the array a[].
- insert():
  1. Prompt the user to input the position (pos) where the element will be inserted.
  2. Prompt the user to enter the new value (data) to be inserted.
  3. Insert the value at position pos-1.
  4. Increment the n(n++).
  5. Display the updated array.
- delete():
  1. Prompt the user to input the position (pos) of the element to be deleted.
  2. Shift elements from pos-1 to the left to overwrite the deleted element.
  3. Decrease the n(n--).
  4. Display the updated array.
- view():
  1. Display all elements in the array.

Step4. Implement the main() function:

- Display a menu with the following options:
  1. CREATE: Call the create() function.
  2. INSERT: Call the insert() function.
  3. DELETE: Call the delete() function.
  4. VIEW: Call the view() function.
  5. QUIT: Exit the program.
- Read the user's choice and perform the corresponding operation using a switch statement.

- Display an error message if the user enters an invalid option.

Step5. Repeat the menu until the user chooses to quit.

Step6. End.

Source code:

```
#include <stdio.h>
#include <stdlib.h>
int a[100],i;
int n=0;
  void create()
   {
        printf("\n Total number of elements in the array : ");
        scanf("%d",&n);
        for(i=0;i < n; i++)
          {
                scanf("%d",&a[i]);
          }
   }
  void insert()
   {
       int pos,i,data;
           printf("\nPosition of insertion : ");
           scanf("%d",&pos);
           for(i=n-1;i>=pos-1;i--)
           {
                a[i+1]=a[i];
           }
        printf("Which value you want to add : ");
        scanf("%d",&data);
        a[pos-1]=data;

        printf("new array is \n");
        n++;
        for(i=0;i < n; i++)
        {
                printf("%d\n",a[i]);
        }
  }

void delete(){
        int pos,i;
        printf("\nEnter the position of array to delete an element : ");
        scanf("%d",&pos);

        for(i=pos-1;i< n-1;i++)
          {
                a[i]=a[i+1];
      }
        printf("New array is : \n ");
```

```
        n--;
        for(i=0;i < n; i++){
        printf("%d\n",a[i]);
        } }

void view(){
        int i;

        printf("Elements are : \n");
        for(i=0;i < n; i++){
        printf("%d\n",a[i]);
        }
}
int main() {
        int opt=0;
        while (1){
                printf("1) CREATE\n");
                printf("2) INSERT\n");
                printf("3) DELETE\n");
                printf("4) VIEW\n");
                printf("5) QUIET\n\n");
                printf("choose your option : ");
                scanf("%d",&opt);
                switch(opt){
                        case 1:
                                create();
                                break;
                        case 2:
                                insert();
                                break;
                        case 3:
                                delete();
                                break;
                        case 4:
                                view();
                                break;
                        case 5: exit(0);
                        default:
                                printf("Invalid option! Try Again.");
                }
        }
        return 0;
}
```

Output:

```
1) CREATE
2) INSERT
3) DELETE
4) VIEW
5) QUIET

choose your option : 1

Total number of elements in the array : 3
11
12
13

1) CREATE
2) INSERT
3) DELETE
4) VIEW
5) QUIET

choose your option : 2

Position of insertion : 1

Which value you want to add : 1
new array is: 1 11 12 13
1) CREATE
2) INSERT
3) DELETE
4) VIEW
5) QUIET

choose your option : 3

Enter the position of array to delete an element : 2
New array is :
 1 12 13
1) CREATE
2) INSERT
3) DELETE
4) VIEW
5) QUIET

choose your option : 4
Elements are :
1 12 13
1) CREATE
2) INSERT
3) DELETE
4) VIEW
5) QUIET

choose your option : 5
```

Experiment No: 4 **Date: 21-11-24**

Aim:

To merge two arrays into one, sort the combined array, and display the result.

Algorithm:

Step1. Start

Step2. Declare arrays a[100] and b[100] for the first and second arrays, respectively.

Step3. Declare variables n1 (size of the first array), n2 (size of the second array), total (total size of the merged array), temp (for swapping), and loop counters i, j.

Step4. Prompt the user to input the size and elements of the first array (a).

Step5. Prompt the user to input the size (n2) and elements of the second array (b).

Step6. Compute the total size of the merged array as total = n1 + n2.

Step7. Append the elements of array b to array a:

- Use a loop to copy elements from b into the empty positions in a starting from index n1.

Step8. Sort the merged array a using Bubble Sort:

- For each element in the array, compare adjacent elements.
- Swap them if the current element is greater than the next element.
- Repeat this process for all elements until the array is sorted.

Step9. Display the sorted merged array.

Step10. End.

Source code:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
        int a[100],b[100];
        int n1,n2,i,temp,total,size=0;

        printf("Enter how much elemenet in first array : ");
        scanf("%d",&n1);
        printf("enter first array");
        for(i=0;i<n1;i++){
                scanf("%d",&a[i]);
        }
        printf("\nEnter how much elemenet in second array : ");
        scanf("%d",&n2);
        printf("enter 2nd array");
        for(i=0;i<n2;i++){
                scanf("%d",&b[i]);
        }
```

```
        total=n1+n2;
        for(i=n1;i<total;i++){
                a[i]=b[size];
                size++; }
        for(i=0;i<total;i++)
        {
                for(int j=0;j<total-1;j++)
                {
                        if(a[j]>a[j+1])
                        {
                                temp=a[j];
                                a[j]=a[j+1];
                                a[j+1]=temp;
                        }
                }}
        printf("Array elements are : \n");
        for(i=0;i<total;i++){
                printf("%d \n",a[i]);
        }
        return 0;
}
```

Output:

```
Enter how much elemenet in first array : 3
enter first array11
12
13

Enter how much elemenet in second array : 3
enter 2nd array4
15
6
Array elements are :
4
6
11
12
13
15


------------------
(program exited with code: 0)
```

Experiment No: 5 **Date: 22-11-24**

Aim:

To implement a menu-driven program in C to perform operations on a singly linked list, including insertion, deletion, searching, and display.

Algorithm:

Step1:Start
Step2: Define a structure node with two fields: data (to store the value) and next
Step3:Initialize head as NULL to represent the empty list.
Step4:Define Operations:
Insertion:
1. Allocate memory for a new node using malloc.
2. Input data for the new node.
3. If the list is empty (head == NULL), make the new node the head and set its next to NULL.
4. Otherwise, point the next of the new node to the current head and update head to the new node.

Delete Operation:
1. Check if the list is empty (head == NULL). If yes, print "List is empty".
2. Otherwise, move head to the next node and free the memory of the previous head node.

Search Operation:
1. Input the value to search.
2. Traverse the list from head using a temporary pointer (temp).
3. Compare the data of each node with the input.
4. If a match is found, print the position and stop searching.
5. If the end of the list is reached without a match, print "Data not found".

Display Operation:
1. Check if the list is empty (head == NULL). If yes, print "List is empty".
2. Otherwise, traverse the list and print the data of each node.

Step5:Main Menu Loop:
Display a menu with the following options: Insert, Delete, Display, Search, and Exit.
Read the user's choice and execute the corresponding operation using a switch statement.
1.Repeat until the user chooses the exit option.
2.Read the user's choice and execute the corresponding operation using a switch statement.
3.Repeat until the user chooses the exit option.
Step6:End

Source code:

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

```
struct node*newnode,*temp,*head=NULL;
 void insert()
 {
        newnode = (struct node*)malloc(sizeof(struct node));
        printf("enter the data: ");
        scanf("%d",&newnode->data);
        if(head==NULL)
          {
                head=newnode;
                head->next = NULL;  }
        else
        {
                newnode->next = head;
                head = newnode;
          }}

 void delete()
  {
     struct node*temp;
     temp=head;
     head=head->next;
     free(temp);
  }
 void search()
 {
        struct node*temp;
        int flag = 0,data,count=1;
        printf("enter the data to be search: ");
        scanf("%d",&data);
        if(head==NULL)
           {
                printf("list is empty.\n");
                return;
           }
           else
         {
                temp=head;
                while(temp->next!=0)
                   {
                        if(data==temp->data)
                           {
                           flag=1;
                           break;    }
                     temp=temp->next;
                     count++;     }
                if(flag==1)
                        printf("the data %d is found at position %d\n",data,count);
                else
                  {
                   printf("the data %d not found",data);
                  }
          }  }
```

```
void display()
  {
        struct node*temp;
        temp=head;
        if(head==NULL)
                printf("list is empty");
        else
          {
                while(temp!=0)
                        {
                                printf(" %d ",temp->data);
                                temp=temp->next;
                        }
         }  }
  int main()
  {
        int opt;
        do
         {
                printf("\n1.insert\n");
                printf("2.deletion \n");
                printf("3.display\n");
                printf("4.search the data\n");
                printf("5.exit\n");
                printf("choose an option ");
                scanf("%d",&opt);

                switch(opt)
                   {
                    case 1: insert();
                          break;
                    case 2: delete();
                          break;
                    case 3: display();
                          break;
                    case 4: search();
                             break;
                    case 5: exit(0);
                    default: printf("invalid option\n");
                   }
         }while(opt!=0);
  }
```

Output:

```
1.insert
2.deletion
3.display
4.search the data
5.exit
choose an option 1
enter the data: 2

1.insert
2.deletion
3.display
4.search the data
5.exit
choose an option 1
enter the data: 3

1.insert
2.deletion
3.display
4.search the data
5.exit
choose an option 1
enter the data: 4

1.insert
2.deletion
3.display
4.search the data
5.exit
choose an option 2

1.insert
2.deletion
3.display
4.search the data
5.exit
choose an option 3
 3  2
1.insert
2.deletion
3.display
4.search the data
5.exit
choose an option 4
enter the data to be search: 2
the data 2 is found at position 2

1.insert
2.deletion
3.display
4.search the data
5.exit
choose an option 5


------------------
(program exited with code: 0)
```

Experiment No: 6 **Date: 25-11-24**

Aim:

To implement a menu-driven program for stack operations (push, pop, peek, and display) using arrays in C.

Algorithm:

step1. Start
step2. Define global variables:
- stack[MAX]: An array to store the elements of the stack.
- top: initialized to `-1`.
- MAX: A constant representing the maximum capacity of the stack.

step3. Define Helper Functions:
- isFull():
  1. Return true if top>MAX-1, indicating the stack is full.
  2. Otherwise, return false.
- isEmpty():
  1. Return true if top == -1, indicating the stack is empty.
  2. Otherwise, return false.

step4. Define Operations:
- push():
  1. Check if the stack is full using isFull().
  2. If yes, print"Stack is full" and exit the function.
  3. Otherwise, prompt the user to input the data.
  4. Increment top and add the data to stack[top].
- pop():
  1. Check if the stack is empty using isEmpty().
  2. If yes, print "Stack is empty" and exit the function.
  3. Otherwise, print the value at `stack[top]` and decrement top.
- peek():
  1. Check if the stack is empty using isEmpty().
  2. If yes, print "Stack is empty".
  3. Otherwise, display the value at stack[top] without modifying top.
- display():
  1. Check if the stack is empty using isEmpty().
  2. If yes, print "Stack is empty".
  3. Otherwise, loop through the stack from 0 to top and display each element.

step5. Main Menu:
1. Use a `while` loop to continuously display the menu with options for Push, Pop, Peek, Display, and Quit.
2. Read the user's choice and call the appropriate function using a `switch` statement.
3. Exit the loop if the user selects the Quit option.

step6. End

Source code:

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

```
int stack[MAX];
 int top=-1,max=100;
 bool isFull()
  {
        if(top>max)
          return 1;
     else
       return 0;
  }
 bool isEmpty()
  {
        if(top<-1)
          return -1;
        else
          return 0;
  }
 void push() {
  int data;
  if(isFull()){
    printf("stack is full");
    return;
     }
  else{
     printf("enter the data");
     scanf("%d",&data);
        top++;
        stack[top]=data;
        printf("\n%d pushed to stack",data);
   }
  }

 void pop()
   {
        if(isEmpty()){
                printf("stack is empty");
                return;
          }
        else  {
                printf("%d is deleted",stack[top]);
                top--;
          }
   }

  void display()
    {
        if(isEmpty()) {
                printf("stack is empty");
                return;
           }
        else {
                for(int i=0;i<=top;i++)
                  {
                        printf("\nstack[%d]=%d",i+1,stack[i]);
```

```
        }
         }   }

void peek()  {
      if(isEmpty())
       {
            printf("stack is empty");
            return;
       }
      else{
            printf("top of the element is %d",stack[top]);
       }
 }
int main()
 {
  int opt;
      while(opt)
       {
            printf("\n1)PUSH\n");
            printf("2)POP\n");
            printf("3)PEEK\n");
            printf("4)DISPLAY\n");
            printf("5)QUIET");
            printf("\nchoose your option : \n");
            scanf("%d",&opt);

            switch(opt)
             {
                  case 1:push();
                         break;
                  case 2:pop();
                         break;
                  case 3:peek();
                         break;
                  case 4:display();
                         break;
                  case 5: exit(0);
                  default: printf("invalid option\n");
             }
            }
      }
```

Output:

```
1)PUSH
2)POP
3)PEEK
4)DISPLAY
5)QUIET
choose your option: 1
enter the data2

2 pushed to stack
1)PUSH
2)POP
3)PEEK
4)DISPLAY
5)QUIET
choose your option: 1
enter the data3

3 pushed to stack
1)PUSH
2)POP
3)PEEK
4)DISPLAY
5)QUIET
choose your option: 2
3 is deleted
1)PUSH
2)POP
3)PEEK
4)DISPLAY
5)QUIET
choose your option: 1
enter the data4

4 pushed to stack
1)PUSH
2)POP
3)PEEK
4)DISPLAY
5)QUIET
choose your option: 3

top of the element is 4
1)PUSH
2)POP
3)PEEK
4)DISPLAY
5)QUIET
choose your option: 4

stack[1]=2
stack[2]=4
1)PUSH
2)POP
3)PEEK
4)DISPLAY
5)QUIET
choose your option: 5


------------------
(program exited with code: 0)
```

Experiment No: 7 **Date: 26-11-24**

Aim:

To implement a queue using an array in C, and provide operations such as enqueue (insertion), dequeue (deletion), display, and quit based on user input.

Algorithm:

Step1. Start
Step2. Define global variables:
- queue[N]: An array of size N to store queue elements.
- front = -1, rear = -1:

Step3. Define Operations:
- enqueue():
  1. Prompt the user for input val.
  2. Check if the queue is full (rear == N-1). If true, print "Queue is overflow" and return.
  3. If the queue is empty (front == -1` and rear == -1), set both `front` and `rear` to 0 and insert the value at queue[rear].
  4. else, increment the rear and insert the value at queue[rear].
- dequeue():
  1. Check if the queue is empty (rear == -1 and front == -1). If true, print "Queue is underflow" and return.
  2. If rear == front, print the value at queue[front] and reset both front&rear=-1.
  3. else, print the value at queue[front] and increment the front to remove the element.
- display():
  1. Check if rear == -1 and front == -1, If true, print "Queue is underflow".
  2. else, iterate through the queue from front to rear and display each element.

Step4. Main Menu:
  1. Display the menu with the following options: enqueue, dequeue, display, and exit.
  2. Use a do-while loop to continually prompt the user for input until the user chooses to exit.
  2. Execute the corresponding operation based on the user's choice using a switch statement.

Step5. End

Source code:

```
#include<stdio.h>
#include<stdlib.h>
#define N 10
int queue[N];
int rear=-1,front=-1;
void enqueue()
  {
  int val;
  printf("enter the data");
  scanf("%d",&val);
  if(rear==N-1) {
          printf("queue is overflow\n");
          return;  }
```

```
 else if(rear==-1 && front==-1)
        {
                rear=front=0;
                queue[rear]=val;
        }
  else{
        rear++;
        queue[rear]=val;
    }
  }

void dequeue()
  {
        if(rear==-1 && front==-1)
         {
                printf("queue is underflow\n");
                return;
         }
        else if(rear==front)
          {
          printf("deleted element is %d",queue[front]);
           rear=front=-1;
          }
        else
          {
                printf("deleted element is %d",queue[front]);
                front++;

          }
  }

 void display()
  {
      int i;
        if(rear==-1 && front==-1)
           {
                printf("queue is underflow\n");
                return;
           }
        else
       {
         for(i=front;i<=rear;i++)
           printf("%d ",queue[i]);
       }
  }
 int main()
  {
        int opt;
        do
          {
```

```
                printf("\n1.enqueue\n");
                printf("2.dequeue\n");
                printf("3.display\n");
                printf("4.exit\n");
                printf("enter the option ");
                scanf("%d",&opt);
                switch(opt)
                  {
                        case 1:enqueue();
                               break;
                        case 2:dequeue();
                               break;
                        case 3:display();
                               break;
                        case 4:exit(0);
                        default:printf("invalid option");
                  }
        }while(1);
}
```

Output:

```
1.enqueue
2.dequeue
3.display
4.exit
enter the option 1
enter the data1

1.enqueue
2.dequeue
3.display
4.exit
enter the option 1
enter the data2

1.enqueue
2.dequeue
3.display
4.exit
enter the option 1
enter the data3

1.enqueue
2.dequeue
3.display
4.exit
enter the option 2
deleted element is 1
1.enqueue
2.dequeue
3.display
4.exit
enter the option 3
2 3
1.enqueue
2.dequeue
3.display
4.exit
enter the option 4


------------------
(program exited with code: 0)
```

Experiment No: 8 **Date: 01-12-24**

Aim:

To implement a stack using a linked list in C, with operations such as push, pop, peek, and display.

Algorithm:

Step1.Start
Step2. Define the structure for a node
- Create a struct node that contains an integer data and a pointer next to the next node in the stack.
- Initialize in global newnode and head pointed to 0

Step3. push():
- Allocate memory for a new node(newnode).
- Prompt the user to enter data to be pushed onto the stack.
- Set the next pointer of the new node to point to the current top of the stack head.
- Update the head pointer to point to the new node
- Print the value that was pushed.

Step4. pop():
- Check if the stack is empty (head == 0), print "stack is empty" and return.
- If the stack is not empty, print head->data.
- Update the head pointer to point to the next node in the stack.
- Free the memory allocated to the node that was popped.

Step5. peek():
- Check if the stack is empty (head == 0), print "stack is empty" and return.
- If the stack is not empty, print head->data.

Step6. display():
- Check if the stack is empty (head == 0), if so, print "stack is empty" and return.
- If the stack has elements, iterate through the stack from head and print each node's data.

Step7 Main Menu:
- Display the following options: push, pop, peek, display, and quit.
- Use a do-while loop to continuously prompt the user for input until the user chooses to exit.
- Execute the corresponding operation based on the user's choice using a switch statement.

Step8.End

Source code:

```
#include<stdio.h>
#include<stdlib.h>
struct node
  {
        int data;
        struct node*next;
  };
 struct node*newnode,*head=0;

  void push() {
        newnode=(struct node*)malloc(sizeof(struct node));

        printf("enter the data : ");
        scanf("%d",&newnode->data);
```

```
        newnode->next=head;
        head=newnode;
        printf("\n%d is pushed into the stack",head->data);
    }

void pop() {
      struct node*temp;

        if(head==0)
          {
                printf("\nstack is empty ");
                return;
          }
        else
           {
                temp=head;
                printf("the deleted data is %d",temp->data);
                head=head->next;
                free(temp);
           }
    }

void peek()
   {
        if(head==0)
          {
                printf("\nstack is empty");
                return;
          }
        else
          printf("last element is %d",head->data);
   }

void display()
    {
        struct node* temp;
        if(head==0)
           {
                printf("stack is empty");
                return;
           }
        else
                {
                    temp=head;
                    while(temp!=0)
                        {
                         printf("%d ",temp->data);
                         temp=temp->next;
                        }
                }     }
```

```
int main()
    {
      int opt;
      do
       {
            printf("\n1)PUSH\n");
                printf("2)POP\n");
                printf("3)PEEK\n");
                printf("4)DISPLAY\n");
                printf("5)QUIET");
                printf("\nchoose your option : \n");
                scanf("%d",&opt);
                switch(opt)
                 {
                  case 1:push();
                         break;
                  case 2:pop();
                         break;
                  case 3:peek();
                         break;
                  case 4:display();
                         break;
                  case 5: exit(0);
                  default: printf("invalid option\n");
          }
        }while(1);
    }
```

Output:

```
1)PUSH
2)POP
3)PEEK
4)DISPLAY
5)QUIET
choose your option : 1
enter the data : 11

11 is pushed into the stack
1)PUSH
2)POP
3)PEEK
4)DISPLAY
5)QUIET
choose your option : 1
enter the data : 12

12 is pushed into the stack
1)PUSH
2)POP
3)PEEK
4)DISPLAY
5)QUIET
choose your option : 1
enter the data : 13

13 is pushed into the stack
1)PUSH
2)POP
3)PEEK
4)DISPLAY
5)QUIET
choose your option : 2
the deleted data is 13
1)PUSH
2)POP
3)PEEK
4)DISPLAY
5)QUIET
choose your option :
3
last element is 12
1)PUSH
2)POP
3)PEEK
4)DISPLAY
5)QUIET
choose your option : 4
12 11
1)PUSH
2)POP
3)PEEK
4)DISPLAY
5)QUIET
choose your option : 5


------------------
(program exited with code: 0)
```

Experiment No: 9 **Date: 10-12-24**

Aim:

To implement a Circular Queue using array in C, with operations such as enqueue, dequeue, search, and display.

Algorithm:

Step1. start
Step1. Initialize the Queue:
- Define the queue array with a fixed size N=5.
- Set both front and rear to -1

Step2. enqueue():
- Prompt the user to enter the data to be added to the queue.
- Check if the queue is empty (front == -1 and rear == -1).
  - If true, set both front and rear to 0 and insert the data at the rear position.
- Check overflow (rear + 1) % N == front,
  - If true, print "queue is overflow" and return.
- Otherwise, increment the rear ((rear + 1) % N) and insert the new data at that position.

Step3. dequeue():
- Check if the queue is empty (front == -1 and rear == -1).
  - print "queue is underflow" and return.
- If front == rear, reset front and rear to -1
- Otherwise, print the data at the front position and increment the front,(front + 1) % N`.

Step4. search():
- Check if the queue is empty (front == -1 and rear == -1).
  - If true, print "queue is underflow" and return.
- Prompt the user to enter the data to search for in the queue.
- Traverse the queue from front to rear, checking each element.
  - If the data is found, print the position of the element in the queue.
  - If not found, print "element is not found".

Step5. display():
- Check if front == -1 and rear == -1.
  - If true, print "queue is underflow" and return.
- Start from the front index and print each element until reaching the rear index.
- Use circular indexing with modulo to print the elements.

Step6. Menu-Driven Interface
- Display the following options Enqueue, Dequeue,Display, Search,and Exit
- Use a `do-while` loop to continuously prompt the user for an option until the user chooses to exit.

Step7. End

Source code:

```
#include<stdio.h>
#include<stdlib.h>
#define N 5
int queue[N];
int rear=-1, front =-1;

void enqueue() {
   int val;
  printf("enter the data");
```

```
 scanf("%d",&val);
 if(rear==-1 && front==-1)
   {
        rear=front=0;
        queue[rear]=val;
   }
 else if((rear+1)%N==front)
   {
        printf("queue is overflow");
        return;
   }
 else
   {
        rear=(rear+1)%N;
        queue[rear]=val;
   }
 }

void dequeue() {
                if(rear==-1 && front==-1)
                        {
                        printf("queue is underflow");
                        return;
                        }
                else if(rear==front)
                        rear=front=-1;
                else{
                        printf("deleted item is %d ",queue[front]);
                        front=(front+1)%N;
                    }
        }

void search()
        {
                int data,flag=0,count=0;
                if(rear==-1 && front==-1)
                        {
                         printf("queue is underflow");
                         return;
                        }
                else
                        {
                         printf("enter the data");
                         scanf("%d",&data);

                         while(front!=rear)
                                {
                                        if(data==queue[front]){
                                                flag=1;
                                                break;
                                                }
```

```
                                        else{
                                                front=(front+1)%N;
                                                count++;
                                            }
                                    }
                                 if(flag==1)
                                   printf("the element is founded at %d position",count);
                                 else
                                   printf("element is not found");
                                }
        }

void display(){
                int i=front;
                if(rear==-1 && front==-1)
                        {
                        printf("queue is underflow");
                        return;
                        }
                else
                        {
                         while(i!=rear)
                                {
                                  printf("%d ",queue[i]);
                                  i=(i+1)%N;
                                }
                                printf("%d ",queue[rear]);
                        }
        }

int main()
  {
        int opt;
        do
       {
                printf("\n1.enqueue\n");
                printf("2.dequeue\n");
                printf("3.display\n");
                printf("4.search\n");
                printf("5.exit\n");
                printf("enter the option ");
                scanf("%d",&opt);

                switch(opt)
                  {
                        case 1:enqueue();
                                break;
                        case 2:dequeue();
                                break;
                        case 3:display();
                                break;
```

```
                case 4:search();
                        break;
                case 5:exit(0);
                default:printf("invalid option");
                }
        }while(1);
}
```

Output:

```
1.enqueue
2.dequeue
3.display
4.search
5.exit
enter the option 1
enter the data11

1.enqueue
2.dequeue
3.display
4.search
5.exit
enter the option 1
enter the data12

1.enqueue
2.dequeue
3.display
4.search
5.exit
enter the option 1
enter the data13

1.enqueue
2.dequeue
3.display
4.search
5.exit
enter the option 1
enter the data14

1.enqueue
2.dequeue
3.display
4.search
5.exit
enter the option 2
deleted item is 11
```

```
1.enqueue
2.dequeue
3.display
4.search
5.exit
enter the option 4
enter the data13
element is not found
1.enqueue
2.dequeue
3.display
4.search
5.exit
enter the option 3
13
1.enqueue
2.dequeue
3.display
4.search
5.exit
enter the option 5


------------------
(program exited with code: 0)
```

Experiment No: 10 **Date: 11-12-24**

Aim:

To implement and demonstrate the operations of the Disjoint Set Union-Find data structure using the following functions

Algorithm

Step1. Start
Step2. - Create two arrays, parent[MAX] and rank[MAX].
- Initialize all elements in the parent array to -1 and rank array to 0.

Step3. Define makeSet():
- Give an argument Element x.
- Check if parent[x] is not -1,then print the element is already in a set.
- Otherwise, assign x as its own parent (parent[x] = x) and set its rank=0.
- Print a message indicating that a set has been created for the element.

Step4. Define findSet():
- Input:Give an argument element x.
- If parent[x] is -1, print that the element is not in any set and return -1.
- If parent[x] != x, find the representative of the set(findSet(parent[x])).
- Return the representative of the set containing x.

Step5. Define unionSets():
- Input: Elements x and y.
- Find the representatives of the sets containing x and y using the findSet() function.
- If either element is not in any set (findSet() returns -1), return without performing the union.
- If the representatives are different, compare their ranks:
  - If rank[repX] > rank[repY], make repX the parent of repY.
  - If rank[repX] < rank[repY], make repY the parent of repX.
  - If ranks are equal, make repX the parent of repY and increase the rank of repX by 1.
- Print a message indicating that the sets containing x and y have been merged.

Step6. Define displaySets():
- Input: Total number of elements n.
- For each element i in the range [1, n]:
  - If parent[i] != -1, find its representative using findSet() and print its representative.

Step7. Define main():
- Initialize the parent array with -1 for all indices.
- Present a menu to the user with the following options:
  - 1. Make Set: Input an element and call the makeSet() function.
  - 2. Find Set: Input an element and call the findSet() function. Display its representative if found.
  - 3. Union Sets: Input two elements and call the unionSets() function to merge their sets.
  - 4. Display Sets: Call the displaySets() function to show all elements and their representatives.
  - 5. Exit: Exit the program.
- Repeat the menu until the user chooses to exit.
Step8. End

Source code

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 50
int parent[MAX];
int rank[MAX];

void makeSet(int x) {
   if (parent[x] != -1) {
      printf("\nElement %d is already in the set\n", x);
      return;
   }
   parent[x] = x;
   rank[x] = 0;
   printf("\nSet created for element %d\n", x);
}
int findSet(int x) {
   if (parent[x] == -1) {
      printf("\nElement %d is not in any set\n", x);
      return -1;
   }
   if (parent[x] != x) {
      parent[x] = findSet(parent[x]);
   }
   return parent[x];
}

void unionSets(int x, int y) {
   int repX = findSet(x);
   int repY = findSet(y);

   if (repX == -1 || repY == -1) return;
   if (repX != repY) {
      if (rank[repX] > rank[repY]) {
         parent[repY] = repX;
      } else if (rank[repX] < rank[repY]) {
         parent[repX] = repY;
      } else {
         parent[repY] = repX;
         rank[repX]++;
      }
      printf("\nSets containing %d and %d have been merged\n", x, y);
   } else {
      printf("\nElements %d and %d are already in the same set\n", x, y);
   }
}
void displaySets(int n) {
   printf("\nElement : Parent\n");
   for (int i = 1; i <= n; i++) {
      if (parent[i] != -1) {
         printf("%d : %d\n", i, findSet(i));
      }
   }
```

```
}
int main() {
    int n = MAX;
    for (int i = 0; i < n; i++) {
        parent[i] = -1;
    }
    int choice, x, y;
    do {
        printf("1. Make Set\n");
        printf("2. Find Set\n");
        printf("3. Union Sets\n");
        printf("4. Display Sets\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("\nEnter the element to create a set: ");
                scanf("%d", &x);
                makeSet(x);
                break;
            case 2:
                printf("\nEnter the element to find its set: ");
                scanf("%d", &x);
                y = findSet(x);
                if (y != -1) {
                    printf("The representative of %d is %d\n", x, y);
                }
                break;
            case 3:
                printf("\nEnter the first element: ");
                scanf("%d", &x);
                printf("Enter the second element: ");
                scanf("%d", &y);
                unionSets(x, y);
                break;
            case 4:
                displaySets(n);
                break;
            case 5:
                exit(0);
            default:
                printf("\nInvalid choice\n");
                break;
        }
    } while (1);
    return 0;
}
```

Output:

```
1. Make Set
2. Find Set
3. Union Sets
4. Display Sets
5. Exit
Enter your choice: 1

Enter the element to create a set: 2

Set created for element 2
1. Make Set
2. Find Set
3. Union Sets
4. Display Sets
5. Exit
Enter your choice: 1

Enter the element to create a set: 3

Set created for element 3
1. Make Set
2. Find Set
3. Union Sets
4. Display Sets
5. Exit
Enter your choice: 1

Enter the element to create a set: 4

Set created for element 4
1. Make Set
2. Find Set
3. Union Sets
4. Display Sets
5. Exit
Enter your choice: 1

Enter the element to create a set: 5

Set created for element 5
1. Make Set
2. Find Set
3. Union Sets
4. Display Sets
5. Exit
```

```
Enter the element to create a set: 5

Set created for element 5
1. Make Set
2. Find Set
3. Union Sets
4. Display Sets
5. Exit
Enter your choice: 3

Enter the first element: 2
Enter the second element: 3

Sets containing 2 and 3 have been merged
1. Make Set
2. Find Set
3. Union Sets
4. Display Sets
5. Exit
Enter your choice: 3

Enter the first element: 4
Enter the second element: 5

Sets containing 4 and 5 have been merged
1. Make Set
2. Find Set
3. Union Sets
4. Display Sets
5. Exit
Enter your choice: 4

Element : Parent
2 : 2
3 : 2
4 : 4
5 : 4

Element 0 is not in any set
50 : -1
1. Make Set
2. Find Set
3. Union Sets
4. Display Sets
5. Exit
Enter your choice: 5


------------------
(program exited with code: 0)
```

Experiment No: 11 **Date: 12-12-24**

Aim:

Implement Kruskal's Algorithm in C to find the Minimum Spanning Tree (MST) of a connected, weighted graph

Algorithm:

Step1. Start
Step2. Input the Number of Vertices:
- Read the number of vertices in the graph (vertex_count).

Step3. Initialize Cost Matrix:
- Input the adjacency matrix representing the graph.
- Replace all 0 values (indicating no edge) with a very high value (999), symbolizing infinity.

Step4. Initialize Parent Array:
- For the Union-Find structure, initialize the `parent` array to track the parent of each vertex.

Step5. Set the Number of Edges in MST:
- The MST will have vertex_count - 1 edges.

Step6. Iteratively Find Minimum Weight Edge:
- For each iteration, find the edge with the smallest weight from the cost matrix that hasn't been included in the MST yet.

Step7. Perform Union-Find Operations:
- Use the Find function to determine the sets of the two vertices connected by the selected edge.
- If the sets are different (no cycle is formed), use the Union function to merge the sets and add the edge to the MST.

Step8. Mark the Selected Edge as Used:
- Update the adjacency matrix to mark the edge as "used" by setting its weight to `999`.

Step9. Repeat Until MST is Formed:
- Continue the process until all edges in the MST (vertex_count - 1 edges) are selected.

Step10. Output the MST:
- Print the edges in the MST and their weights.
- Display the total cost of the MST.

Step11. End

Source code:

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

int parent[MAX];
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;
}

int main(){
	int vertex_count=0;
	int row,column;
	int cost_matrix[MAX][MAX];
	int edge_count=0,count=1;
	int sum_cost=0,min_cost;
	int row_no,column_no,edge1,edge2;

	printf("Implementation of Kruskal's algorithm\n\n");
	printf("Total no of vertex :: ");
	scanf("%d",&vertex_count);

	for(row=1;row<=vertex_count;row++){
		for(column=1;column<=vertex_count;column++){
			scanf("%d",&cost_matrix[row][column]);
			if(cost_matrix[row][column] == 0){
				cost_matrix[row][column] = 999;
			}
		}
	}

	edge_count = vertex_count-1;
	while(count <= edge_count){
		for(row=1,min_cost=999;row<=vertex_count;row++){
			for(column=1;column<=vertex_count;column++){
				if(cost_matrix[row][column] < min_cost){
					min_cost = cost_matrix[row][column];
					edge1 = row_no    =  row;
					edge2 = column_no =  column;
				}
			}
		}
		row_no    = find(row_no);
		column_no = find(column_no);

		if(uni(row_no,column_no)){
			printf("\nEdge %d is (%d -> %d) with cost : %d",
				count++,edge1,edge2,min_cost);
			sum_cost = sum_cost + min_cost;
		}
```

```
            cost_matrix[edge1][edge2] = cost_matrix[edge2][edge1] = 999;
        }
        printf("\n Minimum cost=%d",sum_cost);
        return 0;
}
```

Output:

```
Implementation of Kruskal's algorithm

Total no of vertex :: 3
0
2
3
2
0
1
3
1
0

Edge 1 is (2 -> 3) with cost : 1
Edge 2 is (1 -> 2) with cost : 2
 Minimum cost=3

------------------
(program exited with code: 0)
```

Experiment No: 12 **Date: 13-12-24**

Aim:

To implement Depth-First Search (DFS) traversal of a graph using an adjacency matrix, and display the traversal starting from a specified vertex.

Algorithm:

Step1. Start
Step2. Initialize the Graph:
- Define a 2D array graph[MAX_VERTICES][MAX_VERTICES] to represent the adjacency matrix.
- Define an array visited[MAX_VERTICES] to track whether a vertex has been visited.
- Initialize the number of vertices V.

Step3. Input the Number of Vertices and the Adjacency Matrix
- Ask the user for the number of vertices in the graph.
- For each vertex, take input to fill the adjacency matrix, where graph[i][j] = 1 means there is an edge between vertices i and j, and 0 means no edge exists.

Step4. Implement the DFS Function:
- Define a recursive function dfs(int vertex) to perform DFS starting from the given vertex.
- Mark the current vertex as visited.
- Print the current vertex.
- For each adjacent vertex, if it hasn't been visited and there is an edge between the current vertex and the adjacent vertex, recursively call dfs on the adjacent vertex.

Step5. Traverse the Graph Starting from Vertex 0:
- Initialize the visited array to `0` (not visited) for all vertices.
- Call the dfs function starting from vertex 0.

Step6. Print the DFS traversal result as the program explores the graph.
Step7. End

Source code:

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

int graph[MAX_VERTICES][MAX_VERTICES];
int visited[MAX_VERTICES];
int V;
void dfs(int vertex) {
   visited[vertex] = 1;
   printf("%d ", vertex);
   for (int i = 0; i < V; i++) {
      if (graph[vertex][i] == 1 && !visited[i]) {
         dfs(i);
      }
   }
}
```

```
int main() {
    printf("Enter the number of vertices: ");
    scanf("%d", &V);

    printf("Enter the adjacency matrix (0 for no edge, 1 for edge):\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            scanf("%d", &graph[i][j]);
        }
    }
    for (int i = 0; i < V; i++) {
        visited[i] = 0;
    }
    printf("DFS Traversal starting from vertex 0: ");
    dfs(0);
    return 0;
}
```

Output:

```
Enter the number of vertices: 4
Enter the adjacency matrix (0 for no edge, 1 for edge):
1
0
1
0
0
1
0
0
1
0
0
1
1
1
0
0
DFS Traversal starting from vertex 0: 0 2 3 1

------------------
(program exited with code: 0)
```

Experiment No: 13 **Date: 15-12-24**

Aim:

To implement a Breadth-First Search (BFS) algorithm using a queue.

Algorithm:

Step1. Start
Step2. Input:
- The number of vertices in the graph, n.
- The adjacency matrix representing the graph, where a 1 indicates an edge between two vertices, and a 0 indicates no edge.
- The starting vertex start from where the BFS traversal begins.

Step3. Initialize:
- A visited[MAX] array set to 0.
- A queue[MAX] to store the vertices for BFS traversal.
- front=0 and rear=0

Step4. Enqueue the starting vertex:
- Mark the starting vertex as visited.
- Enqueue the starting vertex into the queue.

Step5. BFS Traversal:
- While the queue is not empty (front < rear), do the following:
- Dequeue a vertex from the front of the queue.
- Print the dequeued vertex.
- For each adjacent vertex (connected with an edge), if it is not visited, mark it as visited and enqueue it.
- Handle overflow if the queue is full and underflow if the queue is empty.

Step6. Queue Overflow Check:
- If the queue's rear index reaches MAX - 1, it means the queue is full. print "Queue Overflow" and exit the BFS traversal.

Step7. Queue Underflow Check:
- If the front index is greater than the rear index,then Print `"Queue Underflow"` and exit the BFS traversal.

Step8. Print the BFS traversal sequence of vertices starting from the given start vertex.
Step9. End

Source code:

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

void bfs(int graph[MAX][MAX], int visited[MAX], int start, int n) {
    int queue[MAX], front = 0, rear = 0;
    visited[start] = 1;
    queue[rear++] = start;

    printf("BFS Traversal: ");
    while (front < rear) {
        int current = queue[front++];
        printf("%d ", current);

        for (int i = 0; i < n; i++) {
            if (graph[current][i] == 1 && !visited[i]) {
                visited[i] = 1;
                queue[rear++] = i;
            }
        }
    }
    printf("\n");
}
int main() {
    int graph[MAX][MAX], visited[MAX] = {0}, n, start;
    printf("Enter the number of vertices: ");
    scanf("%d", &n);
    printf("Enter the adjacency matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
        }
    }
```

```
for (int i = 0; i < n; i++) {
        visited[i] = 0;
    }
    printf("Enter the starting vertex: ");
    scanf("%d", &start);
    bfs(graph, visited, start, n);
    return 0;
}
```

Output:

```
Enter the number of vertices: 4
Enter the adjacency matrix:
0
1
1
0
1
0
1
1
1
0
0
1

0
1
0
1
Enter the starting vertex: 0
BFS Traversal: 0 1 2 3


------------------
(program exited with code: 0)
```

Experiment No: 14 **Date: 18-12-24**

Aim:

To implement the Binary Search Tree with following operations insertion, deletion, and traversal (inorder).

Algorithm:

Step1. Start
Step2. Initialize root as NULL.
Step3. Repeat until user chooses to exit:
  a. Display menu options (Insert, Delete, Display Inorder, Exit).
  b. Read user's choice.
  c. If choice is Insert:
    i. Prompt for data.
    ii. Call insert(root, data)
      -Input: root (the current root of the tree/subtree), data
      -If the current root is NULL, create a new node with the given data and return it.
      -If data is smaller than the current root's data, insert the new data into the left subtree recursively.
      -If data is greater than the current root's data, insert the new data into the right subtree recursively.
  d. If choice is Delete:
    i. Prompt for data.
    ii. Call delete(root, data)
      - Call findMin(struct node* root)
        -Start from the root.
        -Traverse to the leftmost node(minimum value at Right subtree)
      -Return the leftmost node.
  e. If choice is Display Inorder:
    i. Call inorder(root)
      - If the root is NULL, return.
      - Recursively call inorder on the left child.
      - Print the data of the current root.
      - Recursively call inorder on the right child.
  f. If choice is Exit, terminate the program.
Step4. End

Source code:

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

```
  newnode = (struct node*)malloc(sizeof(struct node));
   newnode->data = data;
   newnode->left = newnode->right = NULL;
   return newnode;
}

struct node* insert(struct node* root, int data) {
   if (root == NULL)
      return create(data);
   if (data < root->data) {
      root->left = insert(root->left, data);
   } else {
      root->right = insert(root->right, data);
   }
   return root;
}

void inorder(struct node* root) {
   if (root != NULL) {
      inorder(root->left);
      printf("%d ", root->data);
      inorder(root->right);
   }
}
struct node* search(struct node*root, int data){
          if (root == NULL || root->data == data){
                    return root;
                    }if(data < root->data){
                              return search(root->left,data);
                    }else{
                              return search(root->right,data);
                     }
          }
struct node* findMin(struct node* root) {
   while (root->left != NULL) {
      root = root->left;
   }
   return root;
}

struct node* delete(struct node* root, int data) {
   if (root == NULL)
      return root;
   if (data < root->data) {
      root->left = delete(root->left, data);
   }
   else if (data > root->data) {
      root->right = delete(root->right, data);
   }
   else {
      if (root->left == NULL && root->right == NULL) {
         free(root);
```

```
    return NULL;
    }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* temp = findMin(root->right);
      root->data = temp->data;
      root->right = delete(root->right, temp->data);
    }
  }
  return root;
}

int main() {
  int opt, data;
  do {
    printf("\n1. Insert\n");
    printf("2. Deletion\n");
    printf("3. Display inorder\n");
    printf("4. Exit\n");
    printf("Enter the option: ");
    scanf("%d", &opt);

    switch (opt) {
      case 1:
        printf("Enter the data: ");
        scanf("%d", &data);
        root = insert(root, data);
        printf("Value %d inserted into the BST.\n", data);
        break;
      case 2:
        printf("Enter the data to delete: ");
        scanf("%d", &data);
        root = delete(root, data);
        printf("Value %d deleted from the BST.\n", data);
        break;
      case 3:
        printf("Inorder Traversal: ");
        inorder(root);
        printf("\n");
        break;
      case 4: printf("enter the element for searching: ");
        scanf("%d",&data);
        struct node *found=search(root,data);
```

```
        if(found){
           printf("data %d is founded",data);

        }else
            printf("value not found");
        break;
      case 5: exit(0);
      default: printf("Invalid option..\n");
    }
  } while (1);
  return 0;
}
```

Output:

```
1. Insert
2. Deletion
3. Display inorder
4. Searching
5. Exit
Enter the option: 1
Enter the data: 30
Value 30 inserted into the BST.

1. Insert
2. Deletion
3. Display inorder
4. Searching
5. Exit
Enter the option: 1
Enter the data: 40
Value 40 inserted into the BST.

1. Insert
2. Deletion
3. Display inorder
4. Searching
5. Exit
Enter the option: 1
Enter the data: 40
Value 40 inserted into the BST.

1. Insert
2. Deletion
3. Display inorder
4. Searching
5. Exit
Enter the option: 3
Inorder Traversal: 30 40 40

1. Insert
2. Deletion
3. Display inorder
4. Searching
5. Exit
Enter the option: 2
Enter the data to delete: 50
Value 50 deleted from the BST.
```

```
1. Insert
2. Deletion
3. Display inorder
4. Searching
5. Exit
Enter the option: 4
enter the element for searching: 30
data 30 is founded
1. Insert
2. Deletion
3. Display inorder
4. Searching
5. Exit
Enter the option: 5


------------------
(program exited with code: 0)
```

Experiment No: 15 **Date: 23-12-24**

Aim:

To implement Prim's algorithm to find the Minimum Spanning Tree (MST) of a connected, weighted graph.

Algorithm:

Step1. Start
Step2. Input:
- Total number of vertices vertex_count.
- A list of vertices (vertex identifiers).
- A cost matrix representing the weights of edges between vertices.

Step3. Initialization:
- Create an array visited[MAX] to keep track of which vertices are included in the MST.
- Set visited[1] = 1
- Initialize edge_count = vertex_count - 1
- Initialize sum_cost = 0, count = 1
- Set min_cost to a large number (999 in this case)

Step4. While count <= edge_count:
1. Initialize min_cost = 999 to track the minimum edge cost.
2. For each pair of vertices (row, column) in the cost matrix:
   If cost_matrix[row][column] < min_cost:
      If visited[row] != 0:
         Update min_cost, vertex1, vertex2
3. After finding the minimum cost edge:
   - If either of the vertices (vertex1 or vertex2) is not visited, add the edge to the MST.
   - Update the visited array for the newly included vertex.
   - Add the edge's cost (min_cost) to sum_cost.
4. Set cost_matrix[vertex1][vertex2] and cost_matrix[vertex2][vertex1] to 999 to avoid re-selecting it.

Step5. Termination:
- Once all vertex_count - 1 edges have been added to the MST, stop the process.
- Print the total sum_cost, which represents the minimum cost of the spanning tree.

Step6. End

Source code:

```
#include<stdio.h>
#define MAX 10
int main(){
        int vertex_array[MAX],counter;
        int vertex_count=0;
        int row,column;
        int cost_matrix[MAX][MAX];
        int visited[MAX]={0};
```

```
int edge_count=0,count=1;
int sum_cost=0,min_cost=0;
int row_no,column_no,vertex1,vertex2;

printf("Total no of vertex: ");
scanf("%d",&vertex_count);
printf("\n-- Enter vertex -- \n\n");
for(counter=1;counter<=vertex_count;counter++){
        printf("vertex[%d]: ",counter);
        scanf("%d",&vertex_array[counter]);
}
printf("\n--- Enter Cost matrix of size %d x %d  ---\n\n",vertex_count,vertex_count);
for(row=1;row<=vertex_count;row++){
        for(column=1;column<=vertex_count;column++){
                scanf("%d",&cost_matrix[row][column]);
                if(cost_matrix[row][column] == 0){
                        cost_matrix[row][column] = 999;
                }
        }
}
printf("\n");
visited[1]=1;
edge_count = vertex_count-1;

while(count <= edge_count){
        for(row=1,min_cost=999;row<=vertex_count;row++){
                for(column=1;column<=vertex_count;column++){
                        if(cost_matrix[row][column] < min_cost){

                        if(visited[row] != 0){
                                min_cost = cost_matrix[row][column];
                                vertex1 = row_no = row;
                                vertex2 = column_no = column;
                                }
                        }
                }
        }
```

```
if(visited[row_no] == 0 || visited[column_no] ==0){
                        printf("\nEdge %d is (%d -> %d) with cost : %d
                        ",count++,vertex_array[vertex1],vertex_array[vertex2],min_cost);
                        sum_cost = sum_cost + min_cost;
                        visited[column_no]=1;
                }
                cost_matrix[vertex1][vertex2] = cost_matrix[vertex2][vertex1] = 999;
        }
        printf("\n\nMinimum cost=%d",sum_cost);
        return 0;
}
```

Output:

```
vertex[1]: 1
vertex[2]: 2
vertex[3]: 3
vertex[4]: 4

--- Enter Cost matrix of size 4 x 4  ---

0
2
0
6
2
0
3
8
0
3
0
0
6
8
0
0


Edge 1 is (1 -> 2) with cost : 2
Edge 2 is (2 -> 3) with cost : 3
Edge 3 is (1 -> 4) with cost : 6

Minimum cost=11

------------------
(program exited with code: 0)
```

Experiment No: 16 **Date: 24-12-24**

Aim:

To implement the Topological sorting in C.

Algorithm:

Step1. Start.

Step1. Initialize the Graph:

- Input the number of vertices V and edges E.
- Initialize the graph with V vertices and an adjacency matrix to store edges.
- Initialize an in-degree array where each vertex's in-degree = 0.

Step2. Add Edges:

- For each edge (u, v), mark adj[u][v] = 1 to indicate a directed edge from u to v.
- Increment the in-degree of vertex v by 1 (in_degree[v]++).

Step3. Find Vertices with Zero In-Degree:

- Create a queue and enqueue all vertices whose in-degree is 0.

Step4. Topological Sort:

- While the queue is not empty:
  - Dequeue a vertex u and add it to the topological order list.
  - For each vertex v adjacent to u (where adj[u][v] == 1), reduce the in-degree of v by 1.
  - If the in-degree of v becomes 0, enqueue v.

Step5. Cycle Detection:

- If the number of vertices processed is not equal to V, a cycle exists in the graph, and topological sorting is not possible.

Step6. Output:

- If no cycle is detected, print the topological order of the vertices.

Step8. End

Source code:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_VERTICES 100
struct Graph {
   int V;  // Number of vertices
   int adj[MAX_VERTICES][MAX_VERTICES];
```

```
int in_degree[MAX_VERTICES];
};
void createGraph(struct Graph* graph, int V) {
    graph->V = V;
 for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            graph->adj[i][j] = 0;
        }
        graph->in_degree[i] = 0;
    }
}

void addEdge(struct Graph* graph, int u, int v) {
    graph->adj[u][v] = 1;  // Add edge from u to v
    graph->in_degree[v]++;  // Increment in-degree of vertex v
}

void topologicalSort(struct Graph* graph) {
    int V = graph->V;
    int queue[MAX_VERTICES], front = 0, rear = 0;
    int top_order[MAX_VERTICES];
    int count = 0;

    for (int i = 0; i < V; i++) {
        if (graph->in_degree[i] == 0) {
            queue[rear++] = i;
        }
    }
    while (front < rear) {
        int u = queue[front++];
        top_order[count++] = u;
        for (int v = 0; v < V; v++) {
            if (graph->adj[u][v] == 1) {
                graph->in_degree[v]--;
                if (graph->in_degree[v] == 0) {
                    queue[rear++] = v;
                }
            }
        }
    }
    if (count != V) {
        printf("There exists a cycle in the graph\n");
    } else {
        printf("Topological Sort: ");
        for (int i = 0; i < count; i++) {
```

```
            printf("%d ", top_order[i]);



            }
        printf("\n");
    }
}

int main() {
    int V, E;
    printf("Enter the number of vertices: ");
     scanf("%d", &V);
    printf("Enter the number of edges: ");
    scanf("%d", &E);
    struct Graph graph;
    createGraph(&graph, V);

    printf("Enter the edges (u v) where there is an edge from u to v:\n");
    for (int i = 0; i < E; i++) {
        int u, v;
        scanf("%d %d", &u, &v);
        addEdge(&graph, u, v);
    }
    topologicalSort(&graph);
    return 0;
}
```

Output:

```
Enter the number of vertices: 4
Enter the number of edges: 4
Enter the edges (u v) where there is an edge from u to v:
3 1
3 2
2 0
1 0
Topological Sort: 3 1 2 0


------------------
(program exited with code: 0)
```