Program 4

BFS Program in C using Adjacency Matrix

In this approach, we will use a 2D array to represent the graph. The array will have the size of n x n where n is the number of nodes in the graph. The value of the array at index [i][j] will be 1 if there is an edge between node i and node j and 0 otherwise.

Program/Source Code

Here is source code of the C program to implement bfs using adjacency matrix

```
#include <stdio.h>
 int n, i, j, visited[10], queue[10], front = -1, rear = -1;
int adj[10][10];

void bfs(int v)
{
    for (i = 1; i <= n; i++)
        if (adj[v][i] && !visited[i])
            queue[++rear] = i;
    if (front <= rear)
    {
        visited[queue[front]] = 1;
        bfs(queue[front++]);
    }
}
void main()
{
    int v;
    printf("Enter the number of vertices: ");
    scanf("%d", &n);
    for (i = 1; i <= n; i++)
    {
        queue[i] = 0;
        visited[i] = 0;
    }
    printf("Enter graph data in matrix form:    \n");
```

```
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            scanf("%d", &adj[i][j]);
    printf("Enter the starting vertex: ");
    scanf("%d", &v);
    bfs(v);
    printf("The node which are reachable are:    \n");
    for (i = 1; i <= n; i++)
        if (visited[i])
            printf("%d\t", i);
        else
            printf("BFS is not possible. Not all nodes are reachable");
    return 0;
}
```

Output:

```
Enter the number of vertices: 4
Enter graph data in matrix form:
0 1 1 0
1 0 0 1
1 0 0 1
0 1 1 0
Enter the starting vertex: 2
The node which are reachable are:
1       2       3       4
```

BFS Program in C using Adjacency List

In this approach, we will use an array of linked lists to represent the graph. The array will have the size of n where n is the number of nodes in the graph. The value of the array at index i will be the head of the linked list which will contain all the nodes which are adjacent to node i.

Program/Source Code

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

struct node
{
    int vertex;
    struct node *next;
};

struct node *createNode(int);

struct Graph
{
    int numVertices;
    struct node **adjLists;
    int *visited;
};

struct Graph *createGraph(int vertices)
{
    struct Graph *graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;

    graph->adjLists = malloc(vertices * sizeof(struct node *));
    graph->visited = malloc(vertices * sizeof(int));

    int i;
    for (i = 0; i < vertices; i++)
    {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }
```

```
    return graph;
}
void addEdge(struct Graph *graph, int src, int dest)
{
    struct node *newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}
struct node *createNode(int v)
{
    struct node *newNode = malloc(sizeof(struct node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

void printGraph(struct Graph *graph)
{
    int v;
    for (v = 0; v < graph->numVertices; v++)
    {
        struct node *temp = graph->adjLists[v];
        printf("\n Adjacency list of vertex %d\n ", v);
        while (temp)
        {
            printf("%d -> ", temp->vertex);
            temp = temp->next;
```

```
        }
        printf("\n");
    }
}
void bfs(struct Graph *graph, int startVertex)
{
    struct node *queue = NULL;
    graph->visited[startVertex] = 1;
    enqueue(&queue, startVertex);

    while (!isEmpty(queue))
    {
        printQueue(queue);
        int currentVertex = dequeue(&queue);
        printf("Visited %d ", currentVertex);

        struct node *temp = graph->adjLists[currentVertex];

        while (temp)
        {
            int adjVertex = temp->vertex;

            if (graph->visited[adjVertex] == 0)
            {
                graph->visited[adjVertex] = 1;
                enqueue(&queue, adjVertex);
            }
            temp = temp->next;
        }
    }
int isEmpty(struct node *queue)
{
```

```
    return queue == NULL;
}

void enqueue(struct node **queue, int value)
{
    struct node *newNode = createNode(value);
    if (isEmpty(*queue))
    {
        *queue = newNode;
    }
    else
    {
        struct node *temp = *queue;
        while (temp->next)
        {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}
int dequeue(struct node **queue)
{
    int nodeData = (*queue)->vertex;
    struct node *temp = *queue;
    *queue = (*queue)->next;
    free(temp);
    return nodeData;
}

void printQueue(struct node *queue)
{
    while (queue)
```

```
    {
        printf("%d ", queue->vertex);
        queue = queue->next;
    }
    printf("\n");
}
int main(void)
{
    struct Graph *graph = createGraph(6);
    printf("\nWhat do you want to do?\n");
    printf("1. Add edge\n");
    printf("2. Print graph\n");
    printf("3. BFS\n");
    printf("4. Exit\n");
    int choice;
    scanf("%d", &choice);
    while (choice != 4)
    {
        if (choice == 1)
        {
            int src, dest;
            printf("Enter source and destination: ");
            scanf("%d %d", &src, &dest);
            addEdge(graph, src, dest);
        }
        else if (choice == 2)
        {
            printGraph(graph);
        }
        else if (choice == 3)
 {
            int startVertex;
```

```
            printf("Enter starting vertex: ");
            scanf("%d", &startVertex);
            bfs(graph, startVertex);
        }
        else
        {
            printf("Invalid choice\n");
        }
        printf("What do you want to do?\n");
        printf("1. Add edge\n");
        printf("2. Print graph\n");
        printf("3. BFS\n");
        printf("4. Exit\n");
        scanf("%d", &choice);
    }
    return 0;
}
```

Output:

```
What do you want to do?
1. Add edge
2. Print graph
3. BFS
4. Exit
1
Enter source and destination: 0 1
What do you want to do?
1. Add edge
2. Print graph
3. BFS
4. Exit
1
Enter source and destination: 0 2
```

What do you want to do?

1. Add edge

2. Print graph

3. BFS

4. Exit

1

Enter source and destination: 1 2

What do you want to do?

1. Add edge

2. Print graph

3. BFS

4. Exit

1

Enter source and destination: 2 3

What do you want to do?

1. Add edge

2. Print graph

3. BFS

4. Exit

2

Adjacency list of vertex 0

2 -> 1 ->

Adjacency list of vertex 1

2 -> 0 ->

Adjacency list of vertex 2

3 -> 1 -> 0 ->

Adjacency list of vertex 3

2 ->

Adjacency list of vertex 4

Adjacency list of vertex 5

What do you want to do?

1. Add edge

2. Print graph

3. BFS

4. Exit

3

Enter starting vertex: 0

0

Visited 0 2 1

Visited 2 1 3

Visited 1 3

Visited 3

What do you want to do?

1. Add edge

2. Print graph

3. BFS

4. Exit

4

DFT program using adjacency matrix

```
#include<stdio.h>
#include<conio.h>
int a[20][20], reach[20], n;
void dfs(int v) {
    int i;
    reach[v] = 1;
```

```
    for (i = 1; i <= n; i++)
        if (a[v][i] && !reach[i]) {
            printf("\n %d->%d", v, i);
            dfs(i);
        }
}
int main(int argc, char **argv) {
    int i, j, count = 0;
    printf("\n Enter number of vertices:");
    scanf("%d", &n);
    for (i = 1; i <= n; i++) {
        reach[i] = 0;
        for (j = 1; j <= n; j++)
            a[i][j] = 0;
    }
    printf("\n Enter the adjacency matrix:\n");
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            scanf("%d", &a[i][j]);
    dfs(1);
    printf("\n");
    for (i = 1; i <= n; i++) {
        if (reach[i])
            count++;
    }
    if (count == n)
        printf("\n Graph is connected");
    else
        printf("\n Graph is not connected");
    return 0;
}
```

Output:

Enter number of vertices: 8

Enter the adjacency matrix:

0 1 0 0 0 0 0 1 0

1 0 1 0 0 0 0 1 0

0 1 0 1 0 1 0 0 1

0 0 0 1 0 1 0 0 0

0 0 1 0 1 0 1 0 0

0 0 0 1 0 1 0 1 1

1 1 0 0 0 0 1 0 1

0 0 1 0 0 0 1 1 0

1->2

2->4

4->3

3->6

3->8

8->5

5->7

DFT using adjacency list

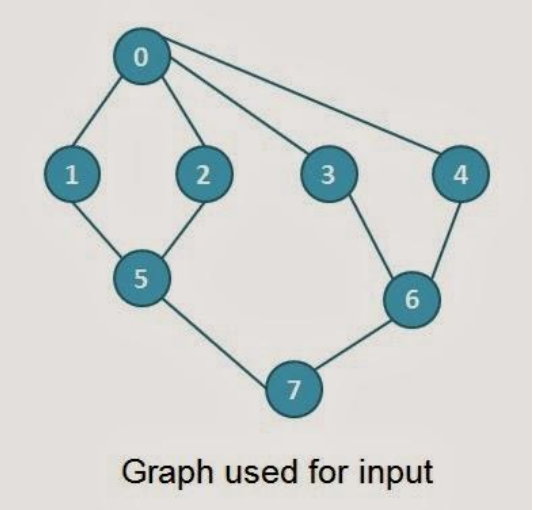
```
#include<stdio.h>
#include<stdlib.h>

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

node *G[20];
//heads of linked list
int visited[20];
int n;
```

```
void read_graph();
//create adjacency list
void insert(int,int);
//insert an edge (vi,vj) in te adjacency list
void DFS(int);

void main()
{
   int i;
   read_graph();
   //initialised visited to 0

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

   DFS(0);
}

void DFS(int i)
{
   node *p;

         printf("\n%d",i);
   p=G[i];
   visited[i]=1;
   while(p!=NULL)
   {
     i=p->vertex;

           if(!visited[i])
         DFS(i);
      p=p->next;
```

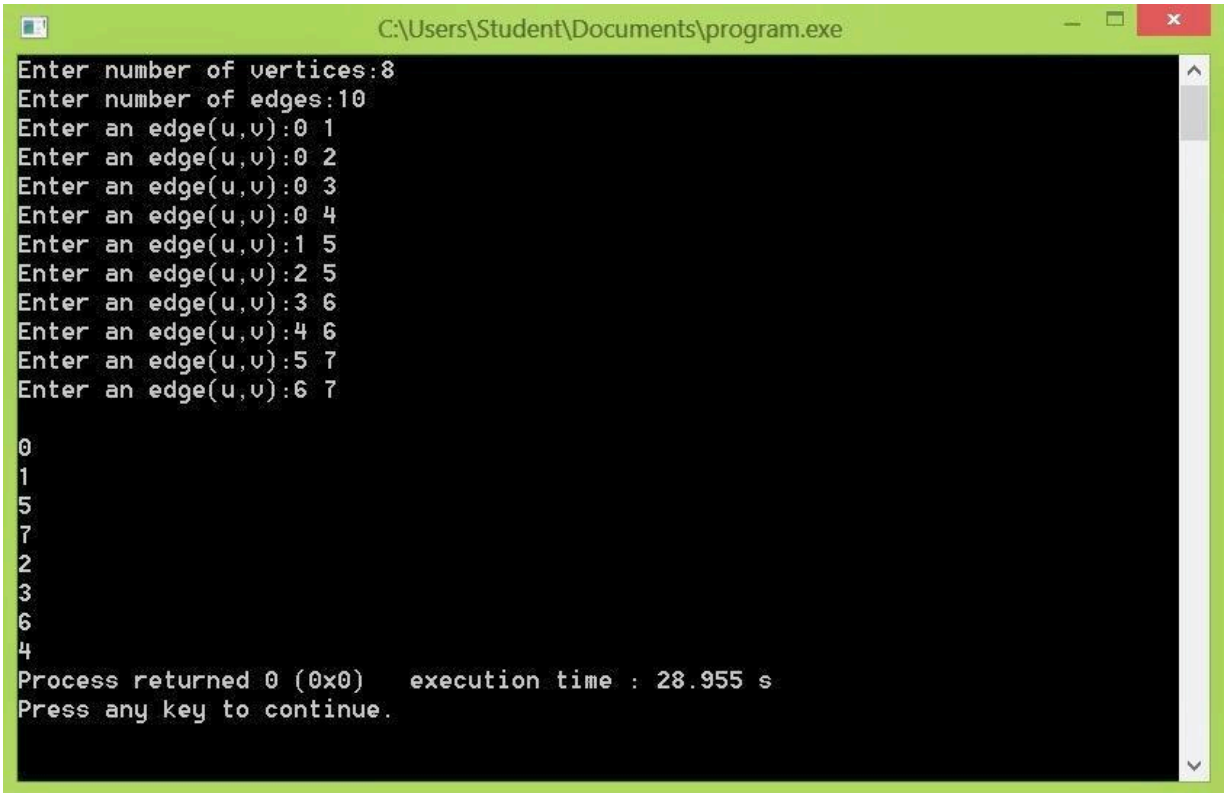
```
    }
}

void read_graph()
{
    int i,vi,vj,no_of_edges;
    printf("Enter number of vertices:");

        scanf("%d",&n);

    //initialise G[] with a null

        for(i=0;i<n;i++)
    {
        G[i]=NULL;
        //read edges and insert them in G[]

                printf("Enter number of edges:");
        scanf("%d",&no_of_edges);

        for(i=0;i<no_of_edges;i++)
        {
        printf("Enter an edge(u,v):");
                        scanf("%d%d",&vi,&vj);
                        insert(vi,vj);
        }
    }
}

void insert(int vi,int vj)
{
    node *p,*q;
```

```
            //acquire memory for the new node
            q=(node*)malloc(sizeof(node));
    q->vertex=vj;
    q->next=NULL;

    //insert the node in the linked list number vi
    if(G[vi]==NULL)
        G[vi]=q;
    else
    {
        //go to end of the linked list
        p=G[vi];

                    while(p->next!=NULL)
            p=p->next;
        p->next=q;
    }
}
```

Example :



Graph used for input



5. **Write a program for finding the bi-connected components in a given graph.**

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

#define NIL -1

// Structure to represent an edge
struct Edge {
    int u, v;
};

// A structure to represent a graph
struct Graph {
```

```
    int V, E; // No. of vertices and edges
    int **adj; // Adjacency matrix
};

// Utility function to create a graph with V vertices
struct Graph* createGraph(int V) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->V = V;
    graph->E = 0;

    graph->adj = (int**)malloc(V * sizeof(int*));
    for (int i = 0; i < V; i++) {
        graph->adj[i] = (int*)calloc(V, sizeof(int));
    }
    return graph;
}

// Utility function to add an edge
void addEdge(struct Graph* graph, int u, int v) {
    graph->adj[u][v] = 1;
    graph->adj[v][u] = 1;
    graph->E++;
}

// Utility function to push an edge into the edge list
void pushEdge(struct Edge* edgeList, int* top, int u, int v) {
    edgeList[*top].u = u;
    edgeList[*top].v = v;
    (*top)++;
}

// Function to find Biconnected Components using DFS traversal
```

```
void BCCUtil(struct Graph* graph, int u, int* disc, int* low, int* parent, struct Edge* edgeList,
int* top, int* time) {
    disc[u] = low[u] = ++(*time);
    int children = 0;


    for (int v = 0; v < graph->V; v++) {
        if (graph->adj[u][v] == 1) {
            if (disc[v] == NIL) {
                children++;
                parent[v] = u;


                pushEdge(edgeList, top, u, v);
                BCCUtil(graph, v, disc, low, parent, edgeList, top, time);


                low[u] = (low[u] < low[v]) ? low[u] : low[v];


                if ((disc[u] == 1 && children > 1) || (disc[u] > 1 && low[v] >= disc[u])) {
                    printf("Biconnected Component: ");
                    while (edgeList[(*top)-1].u != u || edgeList[(*top)-1].v != v) {
                        printf("%d -- %d, ", edgeList[(*top)-1].u, edgeList[(*top)-1].v);
                        (*top)--;
                    }
                    printf("%d -- %d\n", edgeList[(*top)-1].u, edgeList[(*top)-1].v);
                    (*top)--;
                }
            } else if (v != parent[u] && disc[v] < disc[u]) {
                low[u] = (low[u] < disc[v]) ? low[u] : disc[v];
                pushEdge(edgeList, top, u, v);
            }
        }
    }
}
```

```
// The main function to find all Biconnected Components in a given graph
void BCC(struct Graph* graph) {
    int* disc = (int*)malloc(graph->V * sizeof(int));
    int* low = (int*)malloc(graph->V * sizeof(int));
    int* parent = (int*)malloc(graph->V * sizeof(int));
    struct Edge* edgeList = (struct Edge*)malloc(graph->E * sizeof(struct Edge));

    for (int i = 0; i < graph->V; i++) {
        disc[i] = NIL;
        low[i] = NIL;
        parent[i] = NIL;
    }

    int top = 0;
    int time = 0;

    for (int i = 0; i < graph->V; i++) {
        if (disc[i] == NIL) {
            BCCUtil(graph, i, disc, low, parent, edgeList, &top, &time);

            // Print remaining edges from the stack
            if (top > 0) {
                printf("Biconnected Component: ");
                while (top > 0) {
                    printf("%d -- %d, ", edgeList[top-1].u, edgeList[top-1].v);
                    top--;
                }
                printf("\n");
            }
        }
    }
```

```
    free(disc);
    free(low);
    free(parent);
    free(edgeList);
}

int main() {
    int V = 12;
    struct Graph* graph = createGraph(V);

    addEdge(graph, 0, 1);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 3);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 4);
    addEdge(graph, 1, 5);
    addEdge(graph, 0, 6);
    addEdge(graph, 5, 6);
    addEdge(graph, 5, 7);
    addEdge(graph, 5, 8);
    addEdge(graph, 7, 8);
    addEdge(graph, 8, 9);
    addEdge(graph, 10, 11);

    printf("Biconnected components in graph:\n");
    BCC(graph);

    // Free the graph
    for (int i = 0; i < V; i++) {
        free(graph->adj[i]);
    }
```

```
    free(graph->adj);
    free(graph);

    return 0;
}
```

6. Implement Quick sort and Merge sort and observe the execution time for various input sizes (Average, Worst and Best cases).

Quick Sort Program in C

```
#include <stdio.h>

void quick_sort(int[],int,int);
int partition(int[],int,int);

int main()
{
        int a[50],n,i;
        printf("How many elements?");
        scanf("%d",&n);
        printf("\nEnter array elements:");

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

        quick_sort(a,0,n-1);
        printf("\nArray after sorting:");

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

```
	return 0;
}
void quick_sort(int a[],int l,int u)
{
	int j;
	if(l<u)
	{
		j=partition(a,l,u);
		quick_sort(a,l,j-1);
		quick_sort(a,j+1,u);
	}
}

int partition(int a[],int l,int u)
{
	int v,i,j,temp;
	v=a[l];
	i=l;
	j=u+1;

	do
	{
		do
			i++;

		while(a[i]<v&&i<=u);

		do
			j--;
		while(v<a[j]);
```

```
                if(i<j)
                {
                        temp=a[i];
                        a[i]=a[j];
                        a[j]=temp;
                }
        }while(i<j);

        a[l]=a[j];
        a[j]=v;

        return(j);
}
```

Merge Sort

```
#include<stdio.h>

void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);

int main()
{
        int a[30],n,i;
        printf("Enter no of elements:");
        scanf("%d",&n);
        printf("Enter array elements:");

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

        mergesort(a,0,n-1);
```

```
	printf("\nSorted array is :");
	for(i=0;i<n;i++)
		printf("%d ",a[i]);

	return 0;
}

void mergesort(int a[],int i,int j)
{
	int mid;

	if(i<j)
	{
		mid=(i+j)/2;
		mergesort(a,i,mid);		//left recursion
		mergesort(a,mid+1,j); //right recursion
		merge(a,i,mid,mid+1,j);		//merging of two sorted sub-arrays
	}
}

void merge(int a[],int i1,int j1,int i2,int j2)
{
	int temp[50];	//array used for merging
	int i,j,k;
	i=i1;	//beginning of the first list
	j=i2;	//beginning of the second list
	k=0;

	while(i<=j1 && j<=j2)  //while elements in both lists
	{
		if(a[i]<a[j])
```

```
                temp[k++]=a[i++];
            else
                temp[k++]=a[j++];
        }

        while(i<=j1)    //copy remaining elements of the first list
            temp[k++]=a[i++];

        while(j<=j2)    //copy remaining elements of the second list
            temp[k++]=a[j++];

        //Transfer elements from temp[] back to a[]
        for(i=i1,j=0;i<=j2;i++,j++)
            a[i]=temp[j];
}
```