BFS

```
#include<iostream>
#include<bits/stdc++.h>
#include <list>

using namespace std;
struct Graph
{
   int V;    // No. of vertices
   list<int> *adj;
};    // Pointer to an array containing adjacency lists

struct Graph* creategraph(int V)
{
            Graph* graph=new Graph;
   graph->V = V;
  graph->adj = new list<int>[V];
}

void addEdge(struct Graph* graph,int v, int w)
{
   graph->adj[v].push_back(w); // Add w to v's list.
}

bool BFS(struct Graph* graph,int s)
{

 bool *visited = new bool[graph->V];
```

```
for(int i = 0; i < graph->V; i++)
   visited[i] = false;

// Create a queue for BFS
list<int> queue;

// Mark the current node as visited and enqueue it
visited[s] = true;
queue.push_back(s);

// 'i' will be used to get all adjacent vertices of a vertex
list<int>::iterator i;

while(!queue.empty())
{
   // Dequeue a vertex from queue and print it
   s = queue.front();
   cout << s << " ";

   queue.pop_front();

   // Get all adjacent vertices of the dequeued vertex s
   // If a adjacent has not been visited, then mark it visited
   // and enqueue it
   for(i = graph->adj[s].begin(); i !=graph-> adj[s].end(); ++i)
   {

      if(!visited[*i])
      {
```

```
            visited[*i] = true;
            par[*i]=s;
            queue.push_back(*i);
          }
       }
    }

}

// Driver program to test methods of graph class
int main()
{
    // Create a graph given in the above diagram
    bool c;
    struct Graph* graph=creategraph(7);
    addEdge(graph,0,1);
    addEdge(graph,0,2);
   addEdge(graph,1,3);
addEdge(graph,4,1);
addEdge(graph,6,4);
addEdge(graph,5,6);
addEdge(graph,5,2);
addEdge(graph,6,0);

    for(int i=0;i<=6;i++)
    {
            cout<<"the BFS traversal from vertex "<< i << " is\n";
            BFS(graph,i);
            cout<<"\n";
```

```
        }

    return 0;
}
```

DFS

```
#include<iostream>
#include<list>

using namespace std;

struct Graph
{
    int V;    // No. of vertices
    list<int> *adj;
};    // Pointer to an array containing adjacency lists


struct Graph* creategraph(int V)
{
   Graph* graph=new Graph;
    graph->V = V;
    graph->adj = new list<int>[graph->V];
}

void addEdge(struct Graph* graph,int v, int w)
{
   graph->adj[v].push_back(w); // Add w to v's list.
```

```
}

void DFSUtil(struct Graph* graph,int v, bool visited[])
{
   // Mark the current node as visited and print it
   visited[v] = true;
   cout << v << " ";

   // Recur for all the vertices adjacent to this vertex
   list<int>::iterator i;
   for (i = graph->adj[v].begin(); i != graph->adj[v].end(); ++i)
   {
                        if (!visited[*i])
         DFSUtil(graph,*i, visited);
          }
}

// DFS traversal of the vertices reachable from v.
// It uses recursive DFSUtil()
void DFS(struct Graph* graph,int v)
{
   // Mark all the vertices as not visited
   bool *visited = new bool[graph->V];
   for (int i = 0; i < graph->V; i++)
      visited[i] = false;

   // Call the recursive helper function to print DFS traversal
   DFSUtil(graph,v, visited);
}
```

```
int main()
{
   // Create a graph given in the above diagram
   struct Graph* graph=creategraph(4);
   addEdge(graph,0, 1);
   addEdge(graph,0, 2);
   addEdge(graph,1, 2);
   addEdge(graph,2, 0);
   addEdge(graph,2, 3);
   addEdge(graph,3, 3);

   cout << "Following is Depth First Traversal (starting from vertex 2) n";
   DFS(graph,2);

   return 0;
}
```