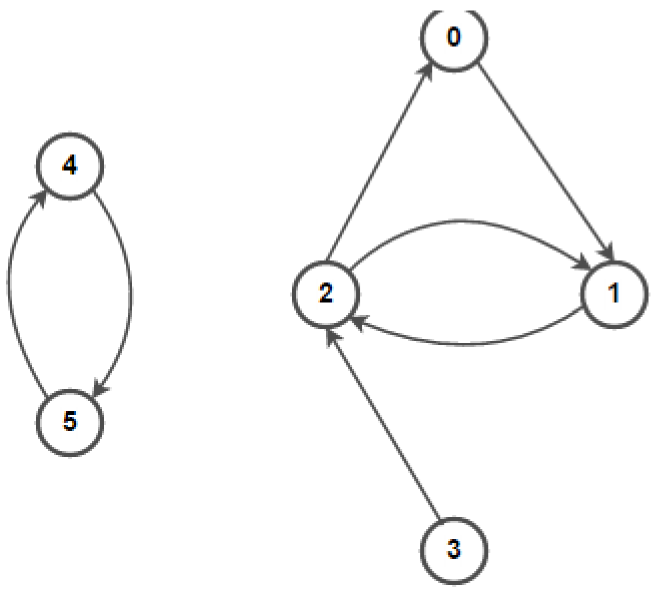
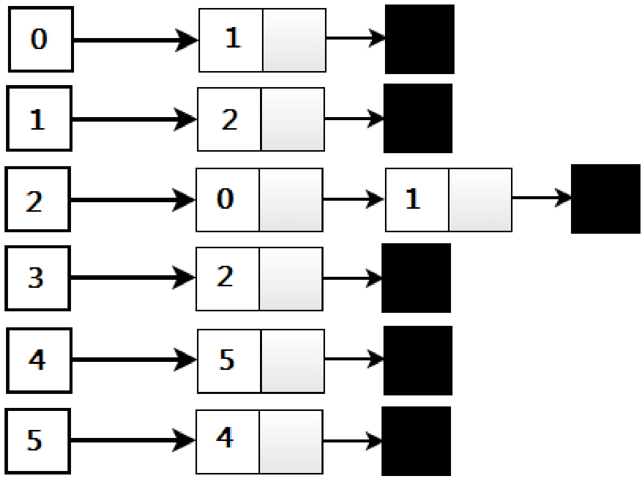
GRAPH

Graph can be implemented using linklist and vectors
for exam you should have understanding of both implementation, also weighted graphs.

For only bfs and dfs you can use STACK/QUEUE library



For example, below is the adjacency list representation of the above graph:



The adjacency list representation of graphs also allows additional data storage on the vertices but is practically very efficient when it contains only a few edges.

C++ CODE WITHOUT USING STL

```
#include <iostream>
using namespace std;

// Data structure to store adjacency list nodes
struct Node
{
   int val;
   Node* next;
};

// Data structure to store a graph edge
struct Edge {
   int src, dest;
};

class Graph
{
   // Function to allocate a new node for the adjacency list
   Node* getAdjListNode(int dest, Node* head)
   {
      Node* newNode = new Node;
      newNode->val = dest;
```

```
        // point new node to the current head
        newNode->next = head;

        return newNode;
    }

    int N;    // total number of nodes in the graph

public:

    // An array of pointers to Node to represent the
    // adjacency list
    Node **head;

    // Constructor
    Graph(Edge edges[], int n, int N)
    {
        // allocate memory
        head = new Node*[N]();
        this->N = N;

        // initialize head pointer for all vertices
        for (int i = 0; i < N; i++) {
            head[i] = nullptr;
        }

        // add edges to the directed graph
        for (unsigned i = 0; i < n; i++)
        {
            int src = edges[i].src;
            int dest = edges[i].dest;

            // insert at the beginning
            Node* newNode = getAdjListNode(dest, head[src]);

            // point head pointer to the new node
            head[src] = newNode;

            // uncomment the following code for undirected graph

            /*
            newNode = getAdjListNode(src, head[dest]);

            // change head pointer to point to the new node
            head[dest] = newNode;
            */
        }
    }
```

```
    // Destructor
    ~Graph() {
        for (int i = 0; i < N; i++) {
            delete[] head[i];
        }

        delete[] head;
    }
};

// Function to print all neighboring vertices of a given vertex
void printList(Node* ptr)
{
    while (ptr != nullptr)
    {
        cout << " —> " << ptr->val;
        ptr = ptr->next;
    }
    cout << endl;
}

// Graph implementation in C++ without using STL
int main()
{
    // an array of graph edges as per the above diagram
    Edge edges[] =
    {
        // pair {x, y} represents an edge from `x` to `y`
        {0, 1}, {1, 2}, {2, 0}, {2, 1}, {3, 2}, {4, 5}, {5, 4}
    };

    // total number of nodes in the graph (labelled from 0 to 5)
    int N = 6;

    // calculate the total number of edges
    int n = sizeof(edges)/sizeof(edges[0]);

    // construct graph
    Graph graph(edges, n, N);

    // print adjacency list representation of a graph
    for (int i = 0; i < N; i++)
    {
        // print given vertex
        cout << i;

        // print all its neighboring vertices
```

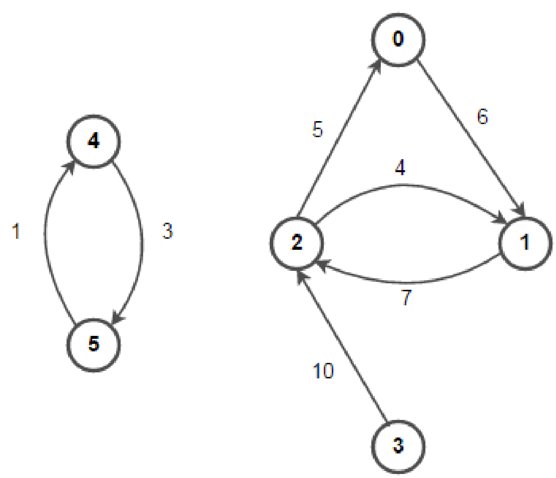
```
    printList(graph.head[i]);
  }

  return 0;
}
```

Output:

```
0 —> 1
1 —> 2
2 —> 1 —> 0
3 —> 2
4 —> 5
5 —> 4
```

WEIGHTED GRAPH C++

We know that in a weighted graph, every edge will have a weight or cost associated with it, as shown below:



Following is the C++ implementation of a directed weighted graph. The implementation is similar to the above implementation of unweighted graphs, except we will also store every edge's weight in the adjacency list.

```
#include <iostream>
using namespace std;

// Data structure to store adjacency list nodes
```

```
struct Node
{
   int val, cost;
   Node* next;
};

// Data structure to store a graph edge
struct Edge {
   int src, dest, weight;
};

class Graph
{
   // Function to allocate a new node for the adjacency list
   Node* getAdjListNode(int value, int weight, Node* head)
   {
      Node* newNode = new Node;
      newNode->val = value;
      newNode->cost = weight;

      // point new node to the current head
      newNode->next = head;

      return newNode;
   }

   int N;    // total number of nodes in the graph

public:

   // An array of pointers to Node to represent the
   // adjacency list
   Node **head;

   // Constructor
   Graph(Edge edges[], int n, int N)
   {
```

```
    // allocate memory
    head = new Node*[N]();
    this->N = N;

    // initialize head pointer for all vertices
    for (int i = 0; i < N; i++) {
        head[i] = nullptr;
    }

    // add edges to the directed graph
    for (unsigned i = 0; i < n; i++)
    {
        int src = edges[i].src;
        int dest = edges[i].dest;
        int weight = edges[i].weight;

        // insert at the beginning
        Node* newNode = getAdjListNode(dest, weight, head[src]);

        // point head pointer to the new node
        head[src] = newNode;

        // uncomment the following code for undirected graph

        /*
        newNode = getAdjListNode(src, weight, head[dest]);

        // change head pointer to point to the new node
        head[dest] = newNode;
        */
    }
}

// Destructor
~Graph() {
    for (int i = 0; i < N; i++) {
        delete[] head[i];
```

```
        }

        delete[] head;
    }
};

// Function to print all neighboring vertices of a given vertex
void printList(Node* ptr, int i)
{
    while (ptr != nullptr)
    {
        cout << "(" << i << ", " << ptr->val << ", " << ptr->cost << ") ";
        ptr = ptr->next;
    }
    cout << endl;
}

// Graph implementation in C++ without using STL
int main()
{
    // an array of graph edges as per the above diagram
    Edge edges[] =
    {
        // (x, y, w) —> edge from `x` to `y` having weight `w`
        {0, 1, 6}, {1, 2, 7}, {2, 0, 5}, {2, 1, 4}, {3, 2, 10}, {4, 5, 1}, {5, 4, 3}
    };

    // total number of nodes in the graph (labelled from 0 to 5)
    int N = 6;

    // calculate the total number of edges
    int n = sizeof(edges)/sizeof(edges[0]);

    // construct graph
    Graph graph(edges, n, N);

    // print adjacency list representation of a graph
```

```
    for (int i = 0; i < N; i++)
    {
        // print all neighboring vertices of a vertex `i`
        printList(graph.head[i], i);
    }

    return 0;
}
```

Output:

```
(0, 1, 6)
(1, 2, 7)
(2, 1, 4)    (2, 0, 5)
(3, 2, 10)
(4, 5, 1)
(5, 4, 3)
```

BFS (ITERATIVE)

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

// Data structure to store a graph edge
struct Edge {
    int src, dest;
};

// A class to represent a graph object
class Graph
{
public:
    // a vector of vectors to represent an adjacency list
    vector<vector<int>> adjList;

    // Graph Constructor
```

```
    Graph(vector<Edge> const &edges, int n)
    {
        // resize the vector to hold `n` elements of type `vector<int>`
        adjList.resize(n);

        // add edges to the undirected graph
        for (auto &edge: edges)
        {
            adjList[edge.src].push_back(edge.dest);
            adjList[edge.dest].push_back(edge.src);
        }
    }
};

// Perform BFS on the graph starting from vertex `v`
void BFS(Graph const &graph, int v, vector<bool> &discovered)
{
    // create a queue for doing BFS
    queue<int> q;

    // mark the source vertex as discovered
    discovered[v] = true;

    // enqueue source vertex
    q.push(v);

    // loop till queue is empty
    while (!q.empty())
    {
        // dequeue front node and print it
        v = q.front();
        q.pop();
        cout << v << " ";

        // do for every edge (v, u)
        for (int u: graph.adjList[v])
        {
```

```
        if (!discovered[u])
        {
            // mark it as discovered and enqueue it
            discovered[u] = true;
            q.push(u);
        }
    }
  }
}

int main()
{
    // vector of graph edges as per the above diagram
    vector<Edge> edges = {
        {1, 2}, {1, 3}, {1, 4}, {2, 5}, {2, 6}, {5, 9},
        {5, 10}, {4, 7}, {4, 8}, {7, 11}, {7, 12}
        // vertex 0, 13, and 14 are single nodes
    };

    // total number of nodes in the graph (labelled from 0 to 14)
    int n = 15;

    // build a graph from the given edges
    Graph graph(edges, n);

    // to keep track of whether a vertex is discovered or not
    vector<bool> discovered(n, false);

    // Perform BFS traversal from all undiscovered nodes to
    // cover all connected components of a graph
    for (int i = 0; i < n; i++)
    {
        if (discovered[i] == false)
        {
            // start BFS traversal from vertex `i`
            BFS(graph, i, discovered);
        }
```

```
    }

    return 0;
}
```

BFS (RECURSION)

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

// Data structure to store a graph edge
struct Edge {
    int src, dest;
};

// A class to represent a graph object
class Graph
{
public:
    // a vector of vectors to represent an adjacency list
    vector<vector<int>> adjList;

    // Graph Constructor
    Graph(vector<Edge> const &edges, int n)
    {
        // resize the vector to hold `n` elements of type `vector<int>`
        adjList.resize(n);

        // add edges to the undirected graph
        for (auto &edge: edges)
        {
            adjList[edge.src].push_back(edge.dest);
            adjList[edge.dest].push_back(edge.src);
        }
    }
};

// Perform BFS recursively on the graph
void recursiveBFS(Graph const &graph, queue<int> &q, vector<bool> &discovered)
{
    if (q.empty()) {
        return;
    }
```

```
    // dequeue front node and print it
    int v = q.front();
    q.pop();
    cout << v << " ";

    // do for every edge (v, u)
    for (int u: graph.adjList[v])
    {
        if (!discovered[u])
        {
            // mark it as discovered and enqueue it
            discovered[u] = true;
            q.push(u);
        }
    }

    recursiveBFS(graph, q, discovered);
}

int main()
{
    // vector of graph edges as per the above diagram
    vector<Edge> edges = {
        {1, 2}, {1, 3}, {1, 4}, {2, 5}, {2, 6}, {5, 9},
        {5, 10}, {4, 7}, {4, 8}, {7, 11}, {7, 12}
        // vertex 0, 13, and 14 are single nodes
    };

    // total number of nodes in the graph (labelled from 0 to 14)
    int n = 15;

    // build a graph from the given edges
    Graph graph(edges, n);

    // to keep track of whether a vertex is discovered or not
    vector<bool> discovered(n, false);

    // create a queue for doing BFS
    queue<int> q;

    // Perform BFS traversal from all undiscovered nodes to
    // cover all connected components of a graph
    for (int i = 0; i < n; i++)
```

```
  {
    if (discovered[i] == false)
    {
      // mark the source vertex as discovered
      discovered[i] = true;

      // enqueue source vertex
      q.push(i);

      // start BFS traversal from vertex `i`
      recursiveBFS(graph, q, discovered);
    }
  }

  return 0;
}
```

DFS

Depth-first search in Graph

A Depth-first search (DFS) is a way of traversing graphs closely related to the preorder traversal of a tree. Following is the recursive implementation of preorder traversal:

```
procedure preorder(treeNode v)
{
    visit(v);
    for each child u of v
        preorder(u);
}
```

To turn this into a graph traversal algorithm, replace "child" with "neighbor". But to prevent infinite loops, keep track of the vertices that are already discovered and not revisit them.

```
procedure dfs(vertex v)
{
    visit(v);
    for each neighbor u of v
        if u is undiscovered
            call dfs(u);
}
```

DFS (RECURSION)

```
#include <iostream>
#include <vector>
using namespace std;

// Data structure to store a graph edge
struct Edge {
   int src, dest;
};

// A class to represent a graph object
class Graph
{
public:
   // a vector of vectors to represent an adjacency list
   vector<vector<int>> adjList;

   // Graph Constructor
   Graph(vector<Edge> const &edges, int n)
   {
      // resize the vector to hold `n` elements of type `vector<int>`
      adjList.resize(n);

      // add edges to the undirected graph
      for (auto &edge: edges)
      {
         adjList[edge.src].push_back(edge.dest);
```

```
            adjList[edge.dest].push_back(edge.src);
        }
    }
};

// Function to perform DFS traversal on the graph on a graph
void DFS(Graph const &graph, int v, vector<bool> &discovered)
{
    // mark the current node as discovered
    discovered[v] = true;

    // print the current node
    cout << v << " ";

    // do for every edge (v, u)
    for (int u: graph.adjList[v])
    {
        // if `u` is not yet discovered
        if (!discovered[u]) {
            DFS(graph, u, discovered);
        }
    }
}

int main()
{
    // vector of graph edges as per the above diagram
    vector<Edge> edges = {
        // Notice that node 0 is unconnected
        {1, 2}, {1, 7}, {1, 8}, {2, 3}, {2, 6}, {3, 4},
        {3, 5}, {8, 9}, {8, 12}, {9, 10}, {9, 11}
    };

    // total number of nodes in the graph (labelled from 0 to 12)
    int n = 13;

    // build a graph from the given edges
    Graph graph(edges, n);

    // to keep track of whether a vertex is discovered or not
    vector<bool> discovered(n);

    // Perform DFS traversal from all undiscovered nodes to
    // cover all connected components of a graph
```

```
    for (int i = 0; i < n; i++)
    {
        if (discovered[i] == false) {
            DFS(graph, i, discovered);
        }
    }

    return 0;
}
```

DFS(ITERATION)

Iterative Implementation of DFS

The non-recursive implementation of DFS is similar to the non-recursive implementation of BFS but differs from it in two ways:

- It uses a stack instead of a queue.
- The DFS should mark discovered only after popping the vertex, not before pushing it.
- It uses a reverse iterator instead of an iterator to produce the same results as recursive DFS.

```
#include <iostream>
#include <stack>
#include <vector>
using namespace std;

// Data structure to store a graph edge
struct Edge {
    int src, dest;
};

// A class to represent a graph object
class Graph
{
public:
    // a vector of vectors to represent an adjacency list
    vector<vector<int>> adjList;

    // Graph Constructor
    Graph(vector<Edge> const &edges, int n)
    {
        // resize the vector to hold `n` elements of type `vector<int>`
        adjList.resize(n);

        // add edges to the undirected graph
```

```
        for (auto &edge: edges)
        {
            adjList[edge.src].push_back(edge.dest);
            adjList[edge.dest].push_back(edge.src);
        }
    }
};

// Perform iterative DFS on graph starting from vertex `v`
void iterativeDFS(Graph const &graph, int v, vector<bool> &discovered)
{
    // create a stack used to do iterative DFS
    stack<int> stack;

    // push the source node into the stack
    stack.push(v);

    // loop till stack is empty
    while (!stack.empty())
    {
        // Pop a vertex from the stack
        v = stack.top();
        stack.pop();

        // if the vertex is already discovered yet,
        // ignore it
        if (discovered[v]) {
            continue;
        }

        // we will reach here if the popped vertex `v` is not discovered yet;
        // print `v` and process its undiscovered adjacent nodes into the stack
        discovered[v] = true;
        cout << v << " ";

        // do for every edge (v, u)
        // we are using reverse iterator (Why?)
        for (auto it = graph.adjList[v].rbegin(); it != graph.adjList[v].rend(); it++)
        {
            int u = *it;
            if (!discovered[u]) {
                stack.push(u);
            }
        }
```

```
    }
}

int main()
{
    // vector of graph edges as per the above diagram
    vector<Edge> edges = {
        // Notice that node 0 is unconnected
        {1, 2}, {1, 7}, {1, 8}, {2, 3}, {2, 6}, {3, 4},
        {3, 5}, {8, 9}, {8, 12}, {9, 10}, {9, 11}
        // {6, 9} introduces a cycle
    };

    // total number of nodes in the graph (labelled from 0 to 12)
    int n = 13;

    // build a graph from the given edges
    Graph graph(edges, n);

    // to keep track of whether a vertex is discovered or not
    vector<bool> discovered(n);

    // Do iterative DFS traversal from all undiscovered nodes to
    // cover all connected components of a graph
    for (int i = 0; i < n; i++)
    {
        if (discovered[i] == false) {
            iterativeDFS(graph, i, discovered);
        }
    }

    return 0;
}
```