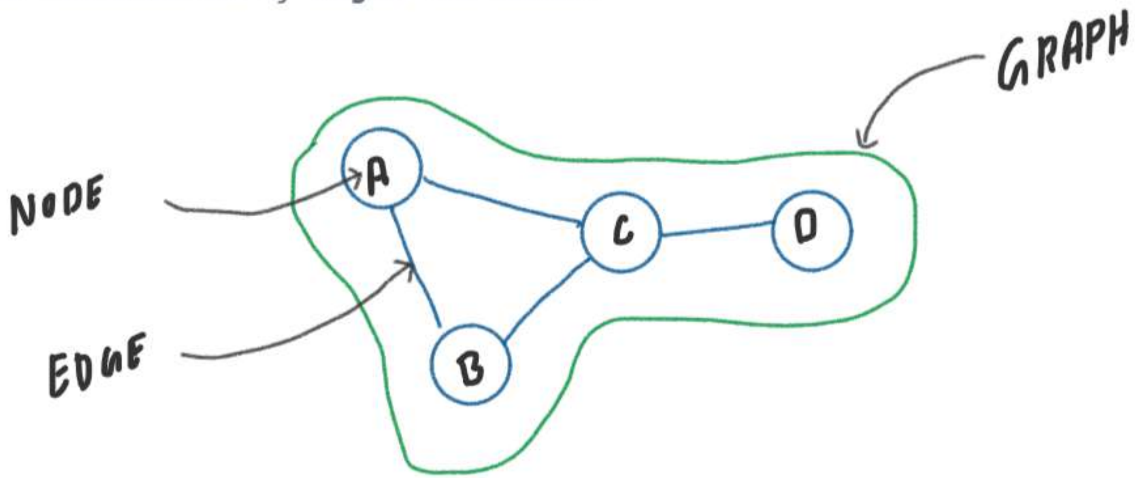
24/01/2024

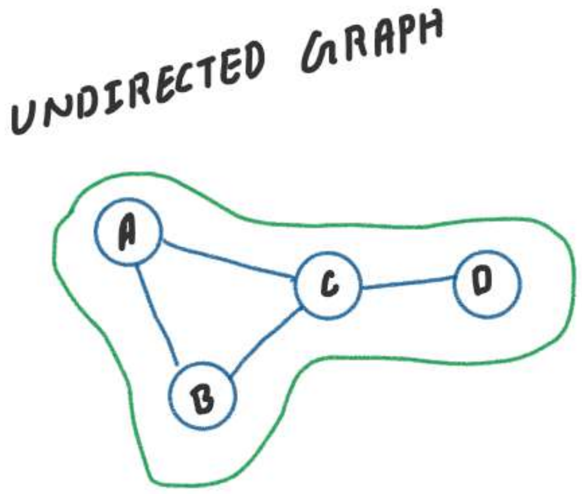
GRAPHS CLASS - 01

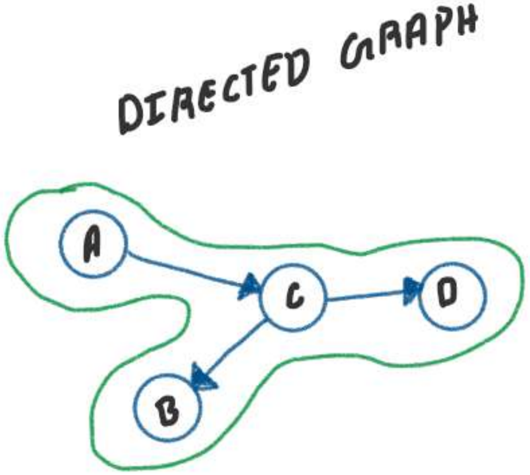
📂 1. What is a Graph?

*It is a **data structure** and combination of **edges** and **nodes**.*



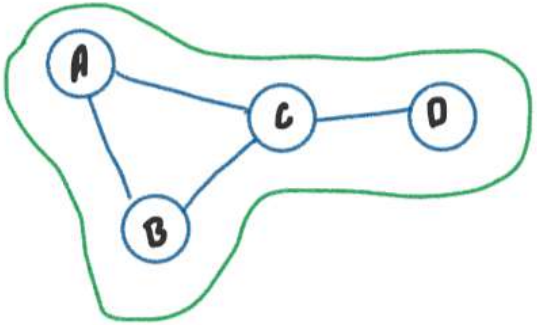
📂 2. Directed and Undirected Graph





📂 3. Edge List

UNDIRECTED GRAPH



UNDIRECTED EDGES LIST

A → C

C → A

A → B

B → A

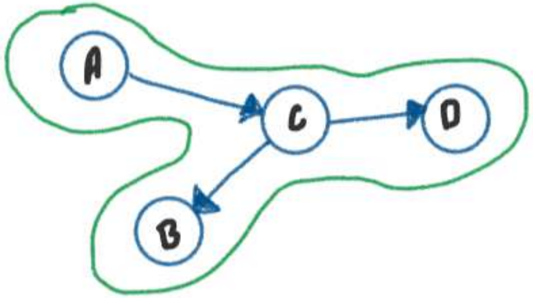
C → B

B → C

C → D

D → C

DIRECTED GRAPH



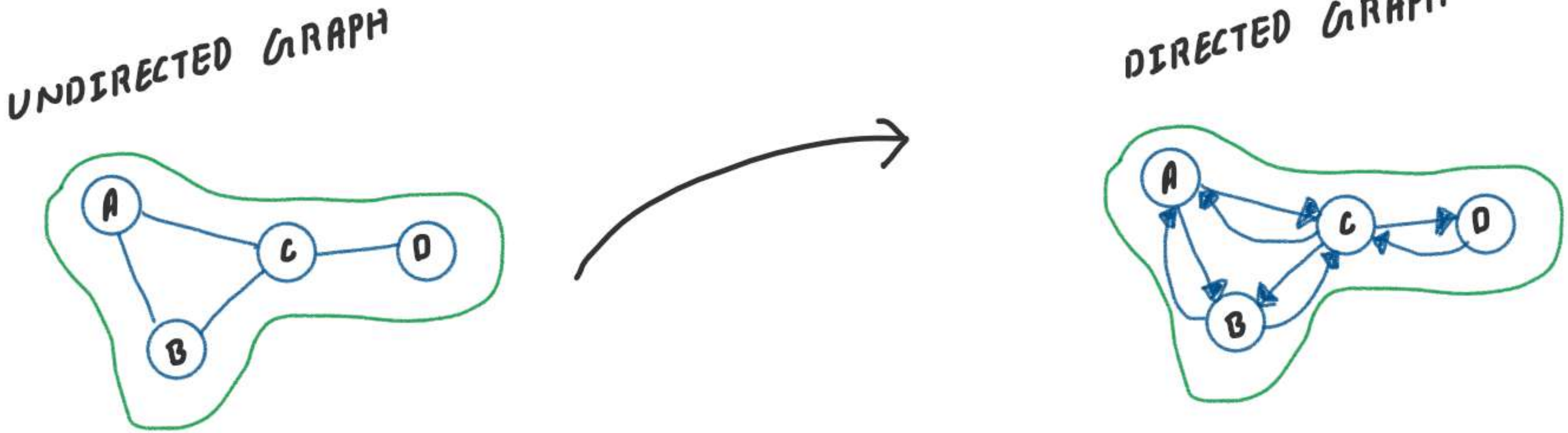
DIRECTED EDGES LIST

A → C

C → B

C → D

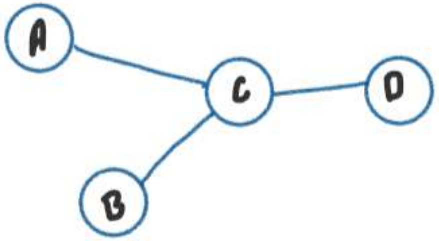
📂 4. Convert from Undirected into Directed



📂 5. Unweighted and Weighted Edge

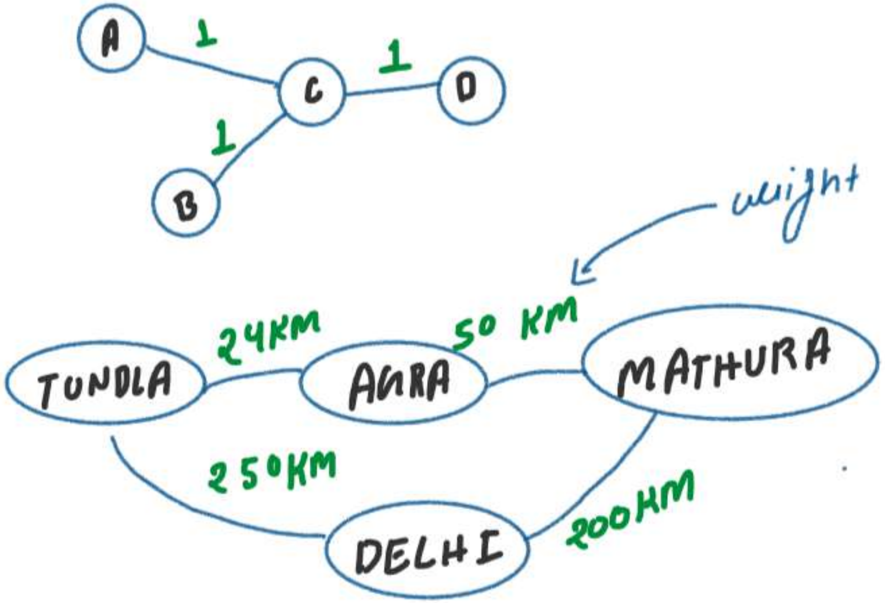
Unweighted graph

Unweighted edge: *edge with no weighted*

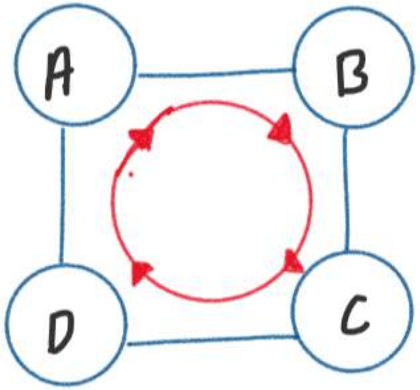


Weighted graph

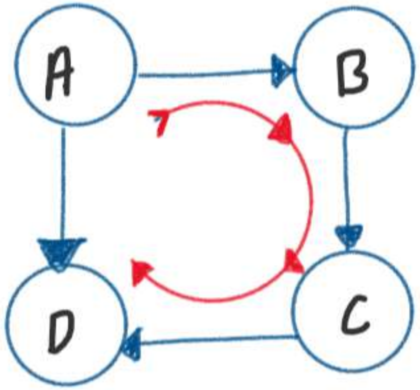
Weighted edge: *edge with weighted*



6. Cyclic and Acyclic Graph

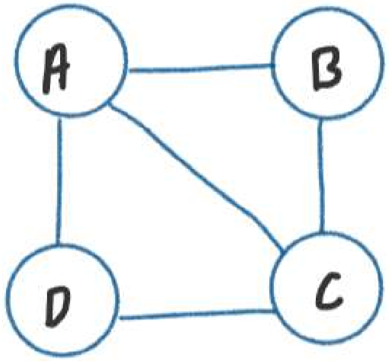


CYCLIC GRAPH

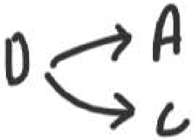


ACYCLIC GRAPH

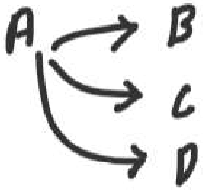
7. Degree



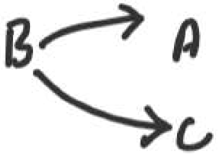
Degree (D) = 2



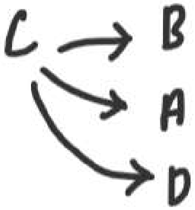
Degree (A) = 3



Degree (B) = 2

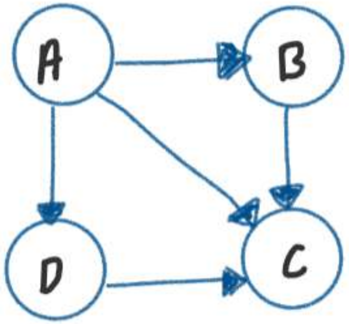


Degree (C) = 3



📂 8. Indegree and Outdegree

Indegree and outdegree are accessed from directed graph always



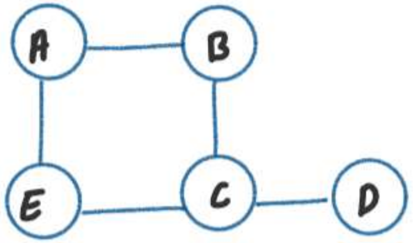
indegree (A) = 0 ← A KI TARAF KON-2 AA RHA HAI

outdegree(A) = 3 → A SE KON-2 DOOR JA RHA HAI

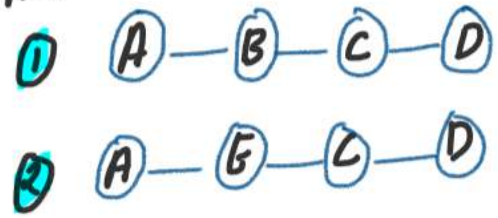
indegree (C) = 3

outdegree (C) = 0

📂 9. Path: Valid and Invalid Path

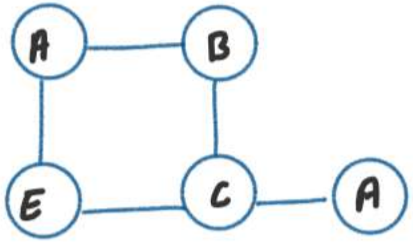


VALID PATHS

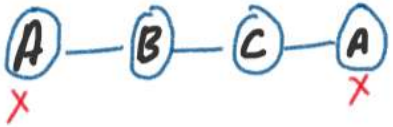


Invalid path:

When any node occurs more than one time in the path.

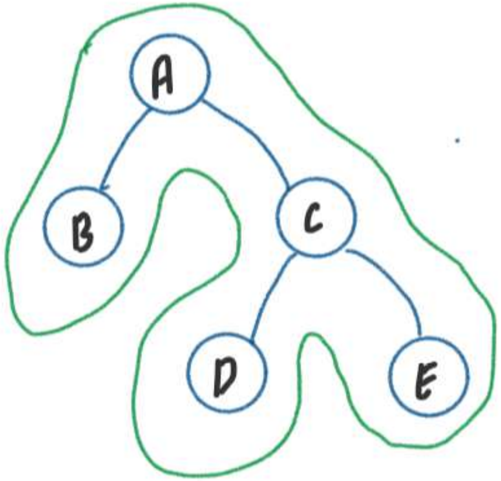


INVALID PATHS

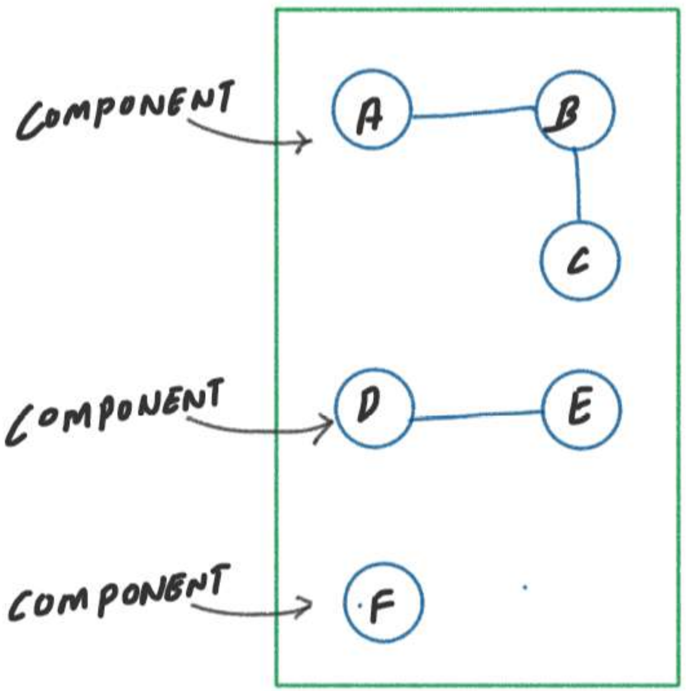


↳ A occurs two times

10. Components (Disconnected and Connected Graph)



CONNECTED GRAPH

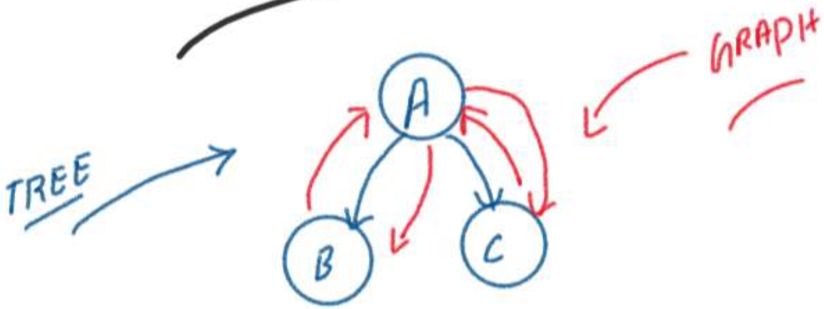


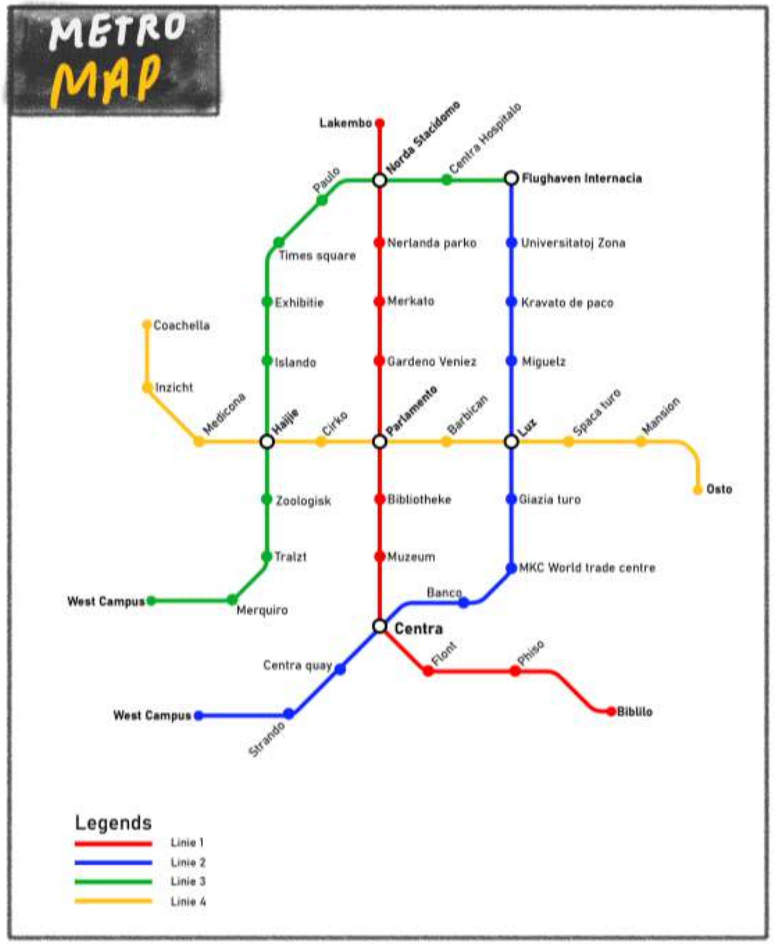
Total components = 3

DISCONNECTED GRAPH

11. Interview Based Question

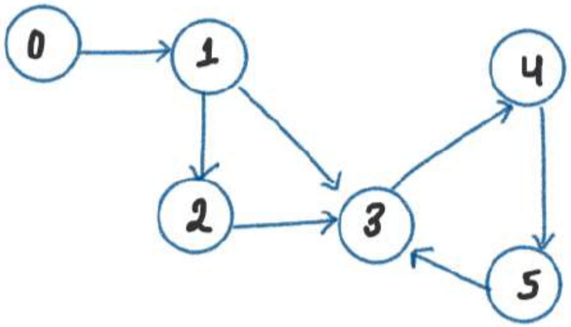
- Practical use of graph
- Is each graph a tree? → NO
- Is each tree a graph? → YES
- Clone a graph → SOLVE ON LEETCODE LATER





📂 12. Graph Creation

***Unweighted** Graph Creation Using Adjacency List*



Directed Graph: 1

READ List

https://en.cppreference.com/w/cpp/container/list

Map<int, List<int>> adjList;

Key	Value List
0	{ 1 }
1	{ 2,3 }
2	{ 3 }
3	{ 4 }
4	{ 5 }
5	{ 3 }

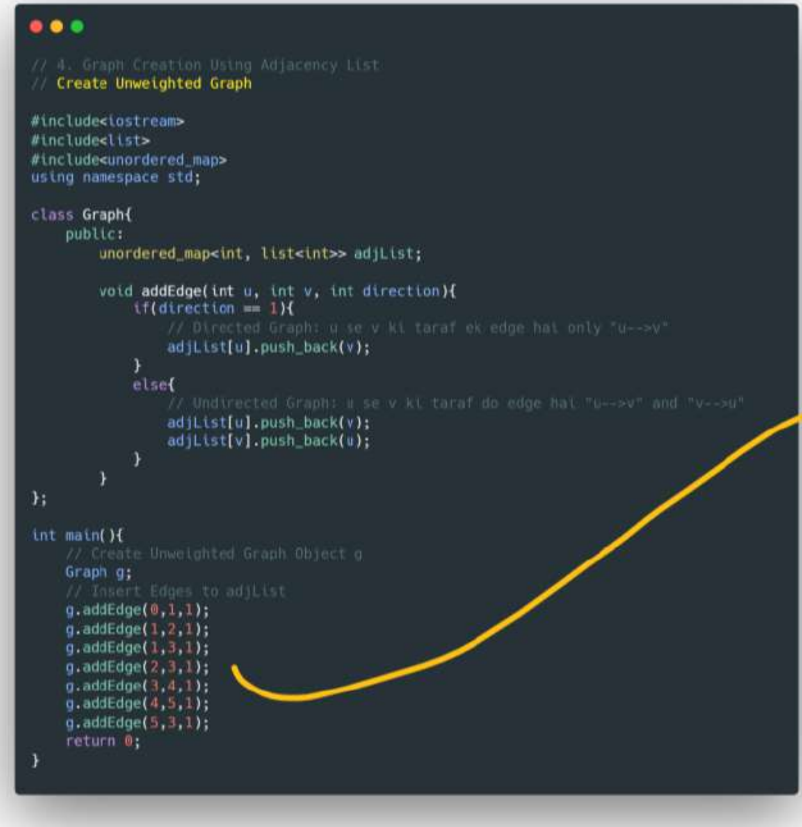
First

Second

Adjacency Edges list

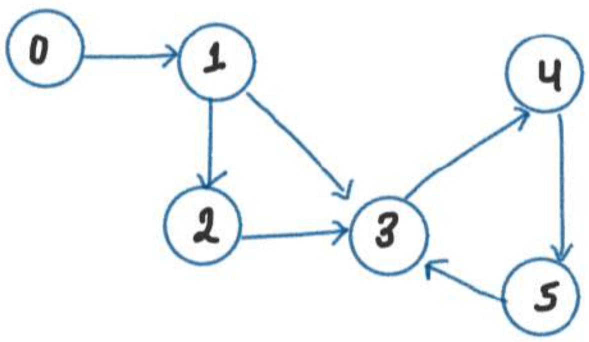
0 → 1
1 → 2
1 → 3
2 → 3
3 → 4
4 → 5
5 → 3

[Note map contains pair]

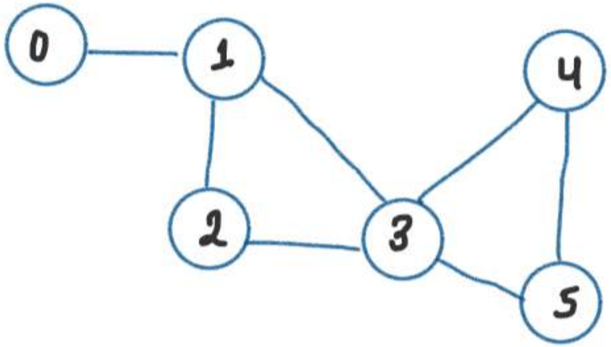


```cpp
// 4. Graph Creation Using Adjacency List
// Create Unweighted Graph

#include<iostream>
#include<list>
#include<unordered_map>
using namespace std;

class Graph{
    public:
        unordered_map<int, list<int>> adjList;

        void addEdge(int u, int v, int direction){
            if(direction == 1){
                // Directed Graph: u se v ki taraf ek edge hai only "u-->v"
                adjList[u].push_back(v);
            }
            else{
                // Undirected Graph: u se v ki taraf do edge hai "u-->v" and "v-->u"
                adjList[u].push_back(v);
                adjList[v].push_back(u);
            }
        }
};

int main(){
    // Create Unweighted Graph Object g
    Graph g;
    // Insert Edges to adjList
    g.addEdge(0,1,1);
    g.addEdge(1,2,1);
    g.addEdge(1,3,1);
    g.addEdge(2,3,1);
    g.addEdge(3,4,1);
    g.addEdge(4,5,1);
    g.addEdge(5,3,1);
    return 0;
}
```

UNWEIGHTED AND DIRECTED GRAPH



Unweighted *Graph Creation Using Adjacency List*

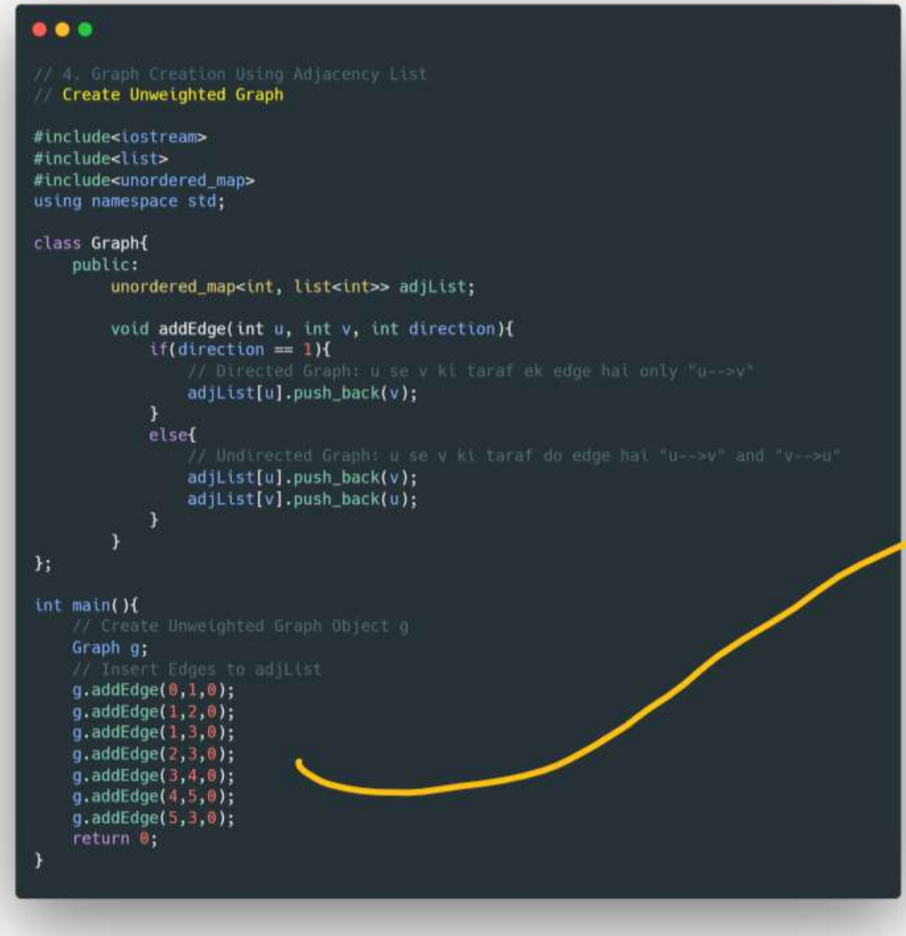


Undirected Graph: 0

Key	Value
0	{1}
1	{0,2,3}
2	{1,3}
3	{1,2,4,5}
4	{3,5}
5	{4,3}

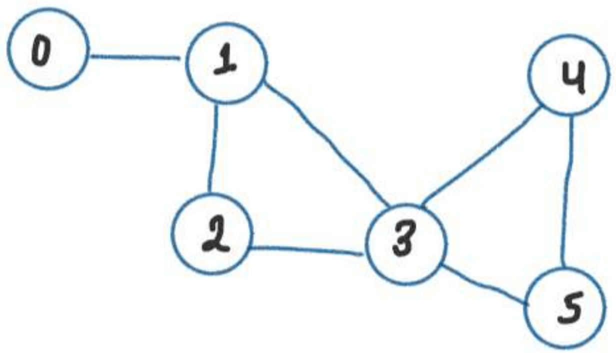
Edges

0 → 1
1 → 0
1 → 2
2 → 1
1 → 3
3 → 1
2 → 3
3 → 2
3 → 4
4 → 3
4 → 5
5 → 4
5 → 3
3 → 5

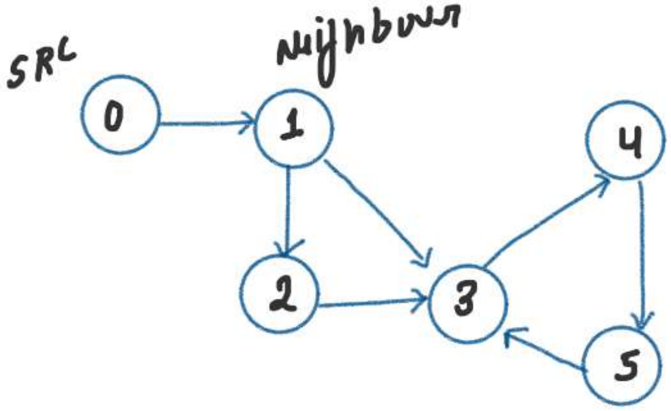


```cpp
// 4. Graph Creation Using Adjacency List
// Create Unweighted Graph

#include<iostream>
#include<list>
#include<unordered_map>
using namespace std;

class Graph{
    public:
        unordered_map<int, list<int>> adjList;

        void addEdge(int u, int v, int direction){
            if(direction == 1){
                // Directed Graph: u se v ki taraf ek edge hai only "u-->v"
                adjList[u].push_back(v);
            }
            else{
                // Undirected Graph: u se v ki taraf do edge hai "u-->v" and "v-->u"
                adjList[u].push_back(v);
                adjList[v].push_back(u);
            }
        }
};

int main(){
    // Create Unweighted Graph Object g
    Graph g;
    // Insert Edges to adjList
    g.addEdge(0,1,0);
    g.addEdge(1,2,0);
    g.addEdge(1,3,0);
    g.addEdge(2,3,0);
    g.addEdge(3,4,0);
    g.addEdge(4,5,0);
    g.addEdge(5,3,0);
    return 0;
}
```

UNWEIGHTED AND UNDIRECTED GRAPH



Unweighted *Graph Creation Using Adjacency List*

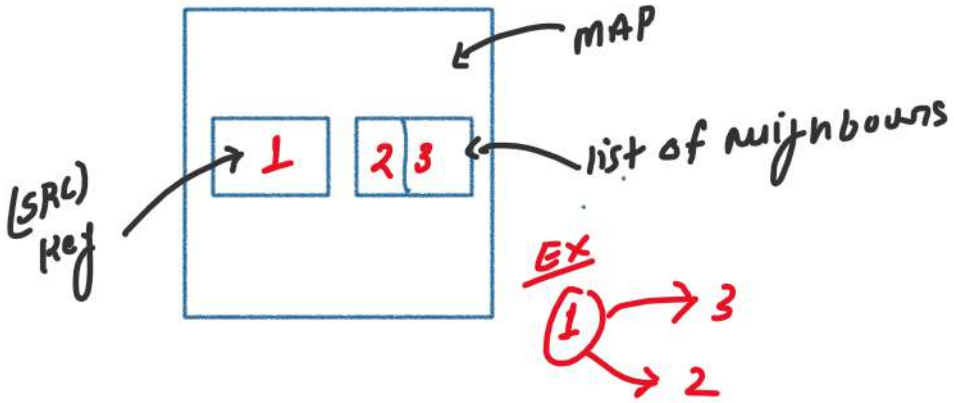


Directed Graph: 1

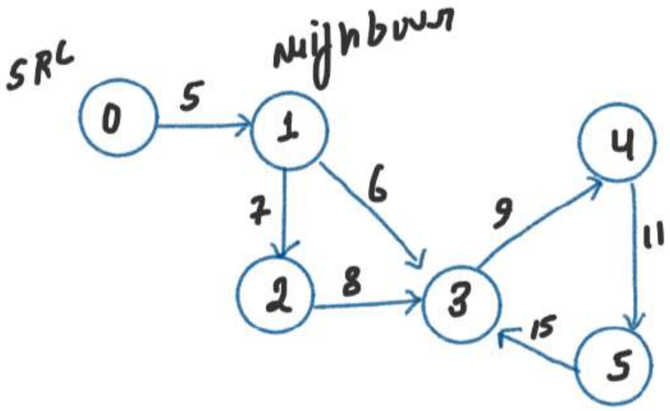
Unweighted Graph

src ↓ neighbours ↓ Edge

```
Map<int, list<int> > adjlist;
```



Weighted Graph Creation Using Adjacency List

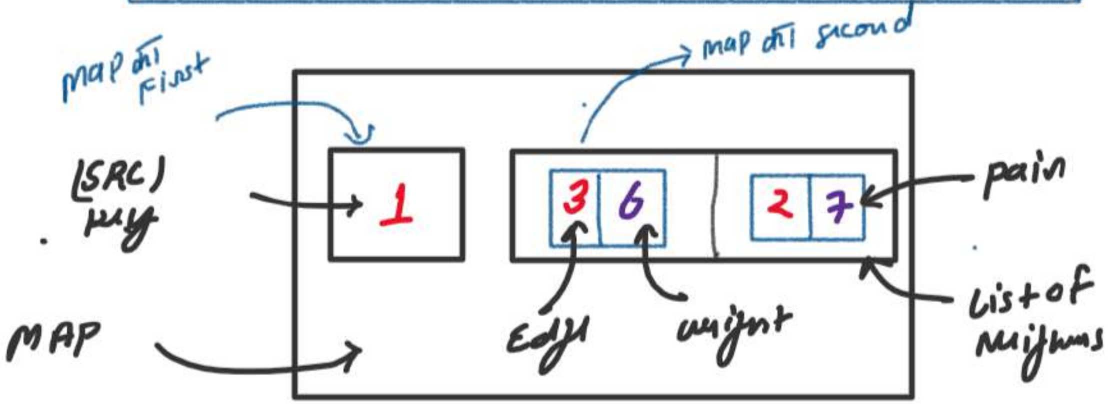


Directed Graph: 1

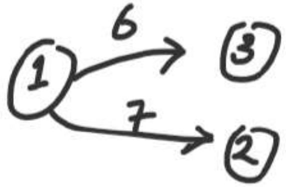
Weighted Graph

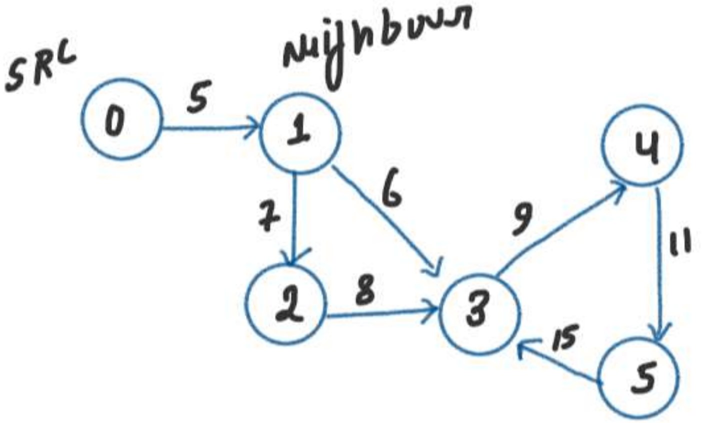
Map< int, list< pair< int, int > >> AdjList;

SRC ↑ (int) · Neigh. ↑ (list) · Edge ↑ (int) · weight ↑ (int)



Ex

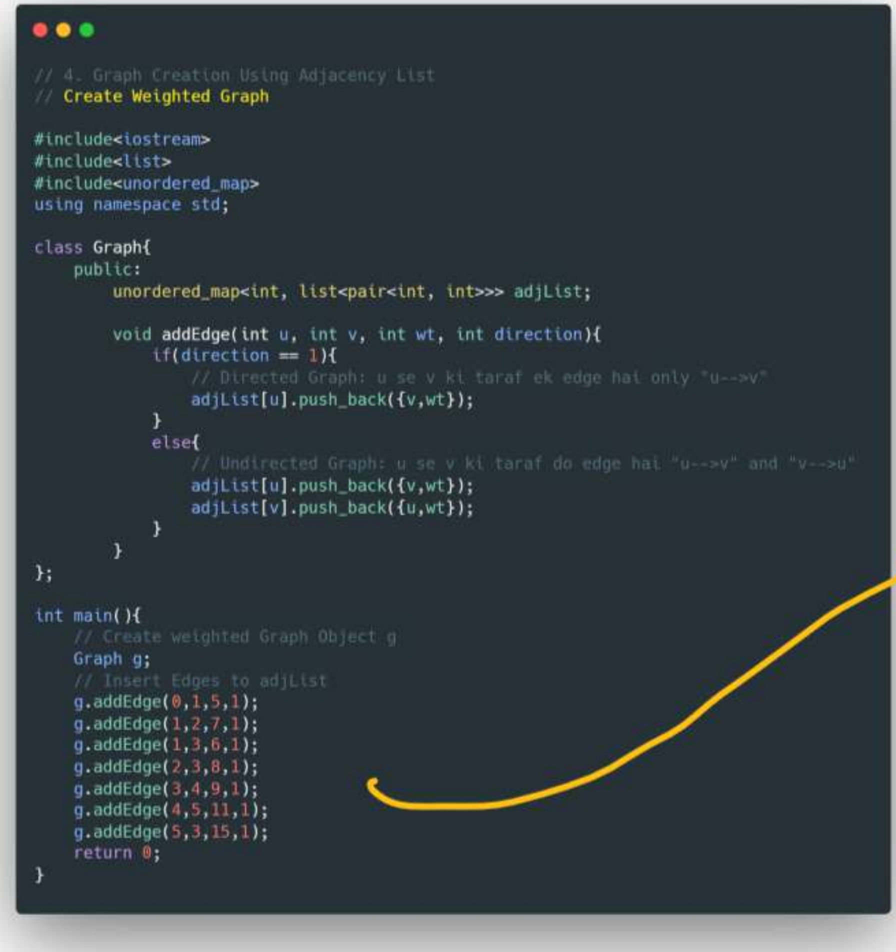




EDGES	WEIGHT
0 → 1	5
1 → 2	7
1 → 3	6
2 → 3	8
3 → 4	9
4 → 5	11
5 → 3	15

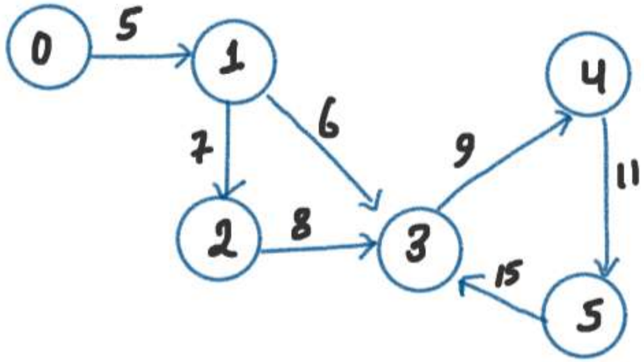
MAP →

Key	Value
0	{ {1, 5} }
1	{ {2, 7}, {3, 6} }
2	{ {3, 8} }
3	{ {4, 9} }
4	{ {5, 11} }
5	{ {3, 15} }

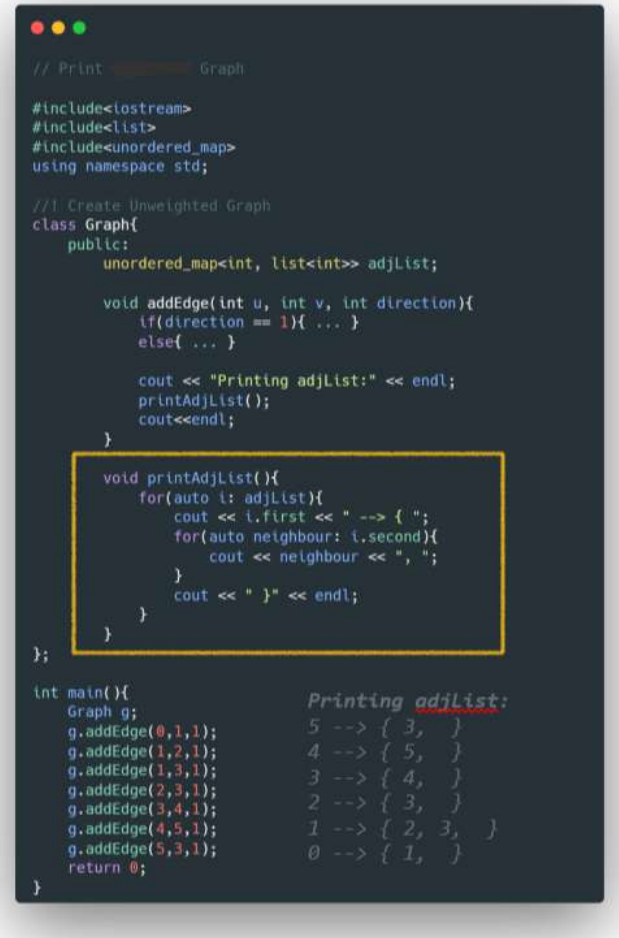


```
// 4. Graph Creation Using Adjacency List
// Create Weighted Graph

#include<iostream>
#include<list>
#include<unordered_map>
using namespace std;

class Graph{
    public:
        unordered_map<int, list<pair<int, int>>> adjList;

        void addEdge(int u, int v, int wt, int direction){
            if(direction == 1){
                // Directed Graph: u se v ki taraf ek edge hai only "u-->v"
                adjList[u].push_back({v,wt});
            }
            else{
                // Undirected Graph: u se v ki taraf do edge hai "u-->v" and "v-->u"
                adjList[u].push_back({v,wt});
                adjList[v].push_back({u,wt});
            }
        }
};

int main(){
    // Create weighted Graph Object g
    Graph g;
    // Insert Edges to adjList
    g.addEdge(0,1,5,1);
    g.addEdge(1,2,7,1);
    g.addEdge(1,3,6,1);
    g.addEdge(2,3,8,1);
    g.addEdge(3,4,9,1);
    g.addEdge(4,5,11,1);
    g.addEdge(5,3,15,1);
    return 0;
}
```

UNWEIGHTED AND DIRECTED GRAPH

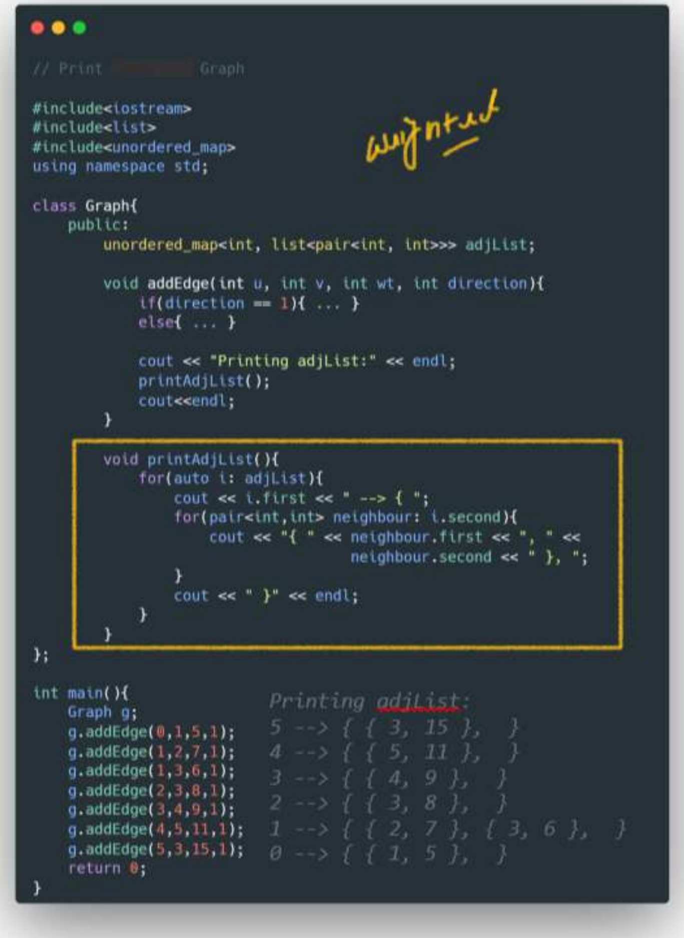


13. Print Graph



```
// Print [redacted] Graph

#include<iostream>
#include<list>
#include<unordered_map>
using namespace std;

//1 Create Unweighted Graph
class Graph{
    public:
        unordered_map<int, list<int>> adjList;

        void addEdge(int u, int v, int direction){
            if(direction == 1){ ... }
            else{ ... }

            cout << "Printing adjList:" << endl;
            printAdjList();
            cout<<endl;
        }

        void printAdjList(){
            for(auto i: adjList){
                cout << i.first << " --> { ";
                for(auto neighbour: i.second){
                    cout << neighbour << ", ";
                }
                cout << " }" << endl;
            }
        }
};

int main(){
    Graph g;
    g.addEdge(0,1,1);
    g.addEdge(1,2,1);
    g.addEdge(1,3,1);
    g.addEdge(2,3,1);
    g.addEdge(3,4,1);
    g.addEdge(4,5,1);
    g.addEdge(5,3,1);
    return 0;
}
```

```
Printing adjList:
5 --> { 3,  }
4 --> { 5,  }
3 --> { 4,  }
2 --> { 3,  }
1 --> { 2, 3,  }
0 --> { 1,  }
```



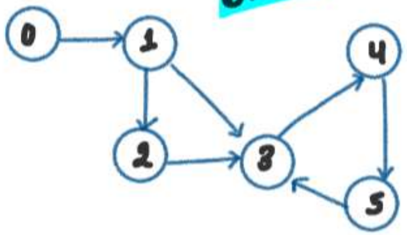
weighted

```
// Print [redacted] Graph

#include<iostream>
#include<list>
#include<unordered_map>
using namespace std;

class Graph{
    public:
        unordered_map<int, list<pair<int, int>>> adjList;

        void addEdge(int u, int v, int wt, int direction){
            if(direction == 1){ ... }
            else{ ... }

            cout << "Printing adjList:" << endl;
            printAdjList();
            cout<<endl;
        }

        void printAdjList(){
            for(auto i: adjList){
                cout << i.first << " --> { ";
                for(pair<int,int> neighbour: i.second){
                    cout << "{ " << neighbour.first << ", " <<
                                    neighbour.second << " }, ";
                }
                cout << " }" << endl;
            }
        }
};

int main(){
    Graph g;
    g.addEdge(0,1,5,1);
    g.addEdge(1,2,7,1);
    g.addEdge(1,3,6,1);
    g.addEdge(2,3,8,1);
    g.addEdge(3,4,9,1);
    g.addEdge(4,5,11,1);
    g.addEdge(5,3,15,1);
    return 0;
}
```

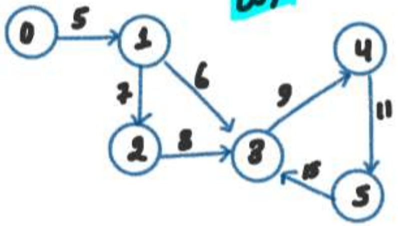
```
Printing adjList:
5 --> { { 3, 15 },  }
4 --> { { 5, 11 },  }
3 --> { { 4, 9 },  }
2 --> { { 3, 8 },  }
1 --> { { 2, 7 }, { 3, 6 },  }
0 --> { { 1, 5 },  }
```

CONNECTED GRAPH

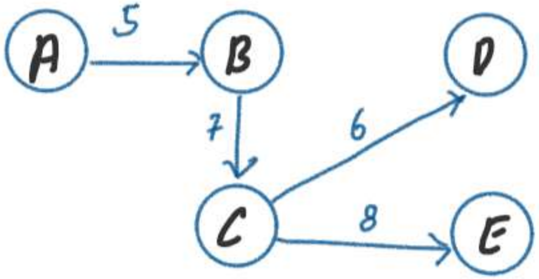
UNWT



WT

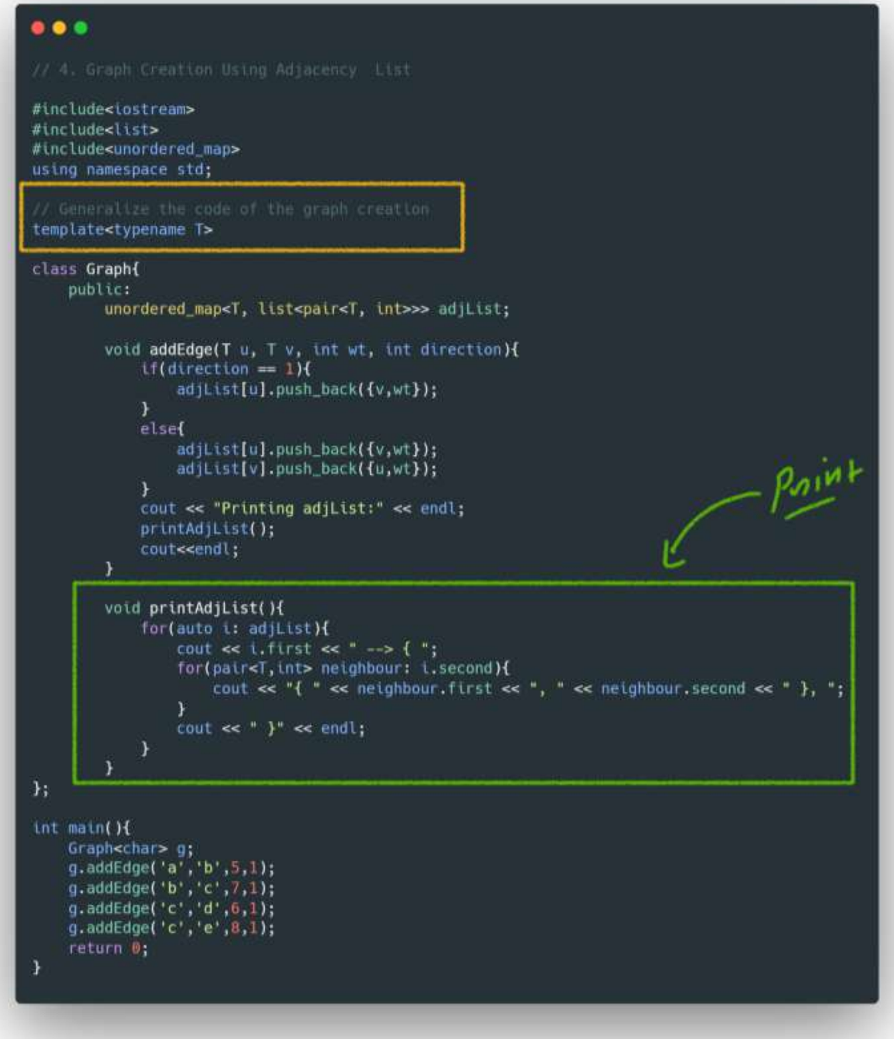


📂 14. Generalize Code of Graph Creation



Output

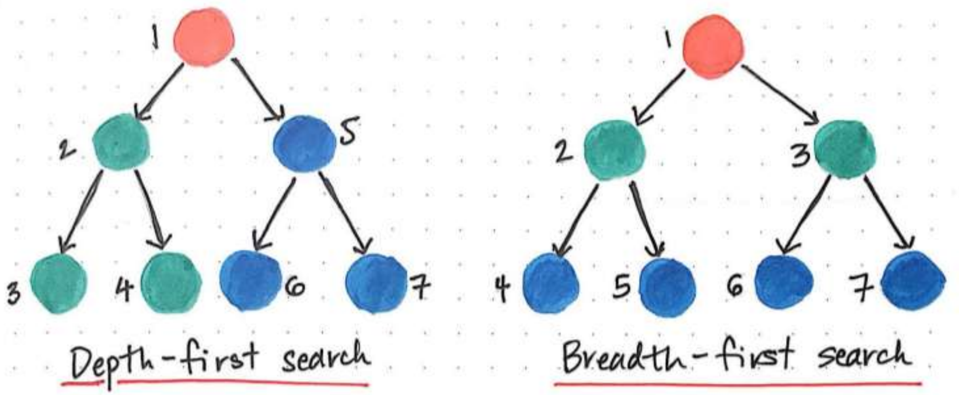
```
Printing adjList:
c --> { { d, 6 }, { e, 8 },  }
b --> { { c, 7 },  }
a --> { { b, 5 },  }
```



```cpp
// 4. Graph Creation Using Adjacency  List

#include<iostream>
#include<list>
#include<unordered_map>
using namespace std;

// Generalize the code of the graph creation
template<typename T>

class Graph{
    public:
        unordered_map<T, list<pair<T, int>>> adjList;

        void addEdge(T u, T v, int wt, int direction){
            if(direction == 1){
                adjList[u].push_back({v,wt});
            }
            else{
                adjList[u].push_back({v,wt});
                adjList[v].push_back({u,wt});
            }
            cout << "Printing adjList:" << endl;
            printAdjList();
            cout<<endl;
        }

        void printAdjList(){
            for(auto i: adjList){
                cout << i.first << " --> { ";
                for(pair<T,int> neighbour: i.second){
                    cout << "{ " << neighbour.first << ", " << neighbour.second << " }, ";
                }
                cout << " }" << endl;
            }
        }
};

int main(){
    Graph<char> g;
    g.addEdge('a','b',5,1);
    g.addEdge('b','c',7,1);
    g.addEdge('c','d',6,1);
    g.addEdge('c','e',8,1);
    return 0;
}
```

📂 15. Traverse the Graph

- **BFS:** *Breath First Search Algorithm*
- **DFS:** *Depth First Search Algorithm*



Depth-first search

Breadth-first search

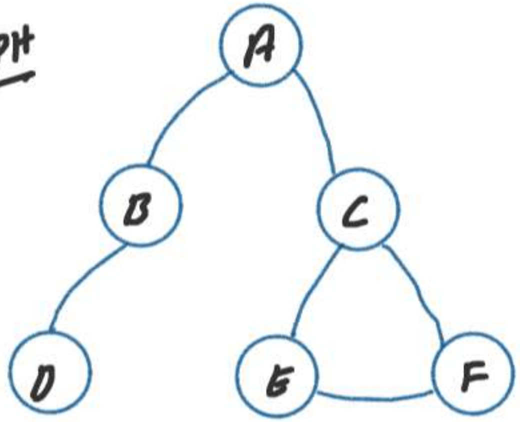
1 → 2 → 3 → 4 → 5 → 6 → 7

1 → 2 → 3 → 4 → 5 → 6 → 7

16. Traverse Graph Using BFS

↳ TRAVERSE Like Level ORder

GRAPH



A's child ⇒ B, C

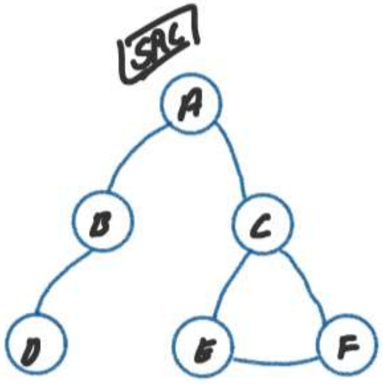
B's child ⇒ D

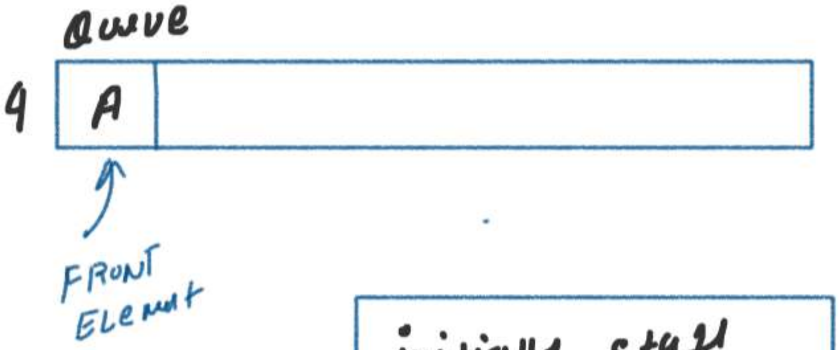
C's child ⇒ E, F

Logic Build

Output

↳ A B C D E F





Adjlist

Key	Value
A	{ B, C }
B	{ D }
C	{ E, F }
D	{ B }
E	{ C, F }
F	{ C, E }

initially stage

q.push(A);

visited[A] = T

visited

Key	Value
A	T
B	F
C	F
D	F
E	F
F	F

STEP1 save front and print and pop

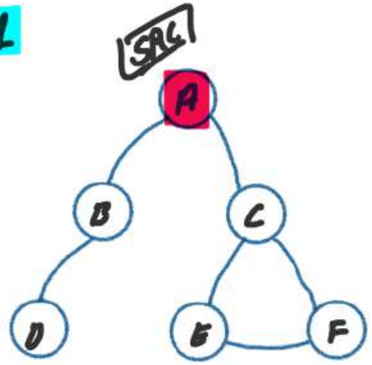
STEP2 go to Adjlist to get the child of front

STEP3 go to visited to check the child is visited or not

→ NOT → q.push(child)
visited [child] = T

→ visited → Ignore kardo

Iteration1



Queue

q

~~A~~	B	C	

FRONT Element

Cout

A

AdjList

Key	Value
A	{ B, C }
B	{ D }
C	{ E, F }
D	{ B }
E	{ C, F }
F	{ C, E }

Visited

Key	Value
A	T
B	~~F~~ T
C	~~F~~ T
D	F
E	F
F	F

use to track the visited child

STEP1 front = A

q.pop() → A

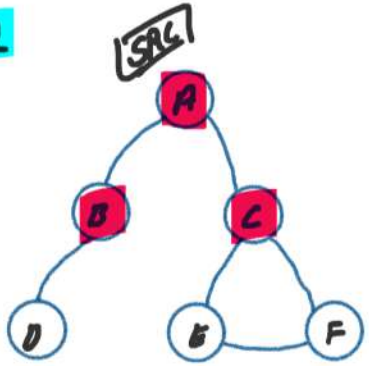
STEP2 {B, C}

STEP3

B → q.push(B)
B → visited[B] = T

C → q.push(C)
C → visited[C] = T

Iteration2



Queue

q	~~B~~	C	D

FRONT Element

Adjlist

Key	Value
A	{ B, C }
B	{ D }
C	{ E, F }
D	{ B }
E	{ C, F }
F	{ C, E }

Cout

A B

Visited

Key	Value
A	T
B	~~F~~ T
C	~~F~~ T
D	~~F~~ T
E	F
F	F

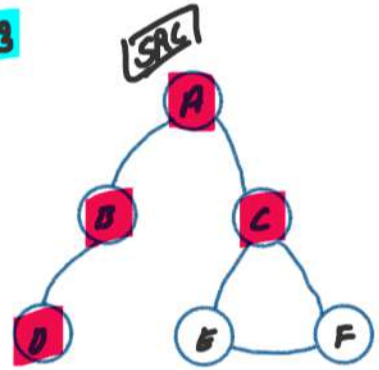
used to track the visited child

STEP1 front = B
q.pop()

STEP2 { D }

STEP3 D → q.push(D)
→ visited[D] = T

Iterations



Queue

q	~~C~~	D	E	F

FRONT Element

Cout

A B C

Adjlist

Key	Value
A	{ B, C }
B	{ D }
C	{ E, F }
D	{ B }
E	{ C, F }
F	{ C, E }

Visited

Key	Value
A	T
B	~~F~~ T
C	~~F~~ T
D	~~F~~ T
E	~~F~~ T
F	~~F~~ T

use to track the visited child

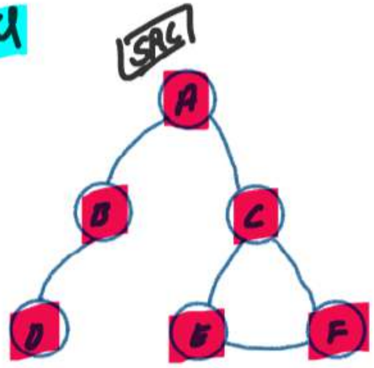
STEP1 front = C
q.pop()

STEP2 { E, F }

STEP3 E → q.push(E)
E → visited[E] = T

F → q.push(F)
F → visited[F] = T

Iteration 4



Queue

q	~~D~~	E	F	

FRONT Element

AdjList

Key	Value
A	{ B, C }
B	{ D }
C	{ E, F }
D	{ B }
E	{ C, F }
F	{ C, E }

Cout

A B C D

STEP1 front = D
q.pop()

STEP2 { B }

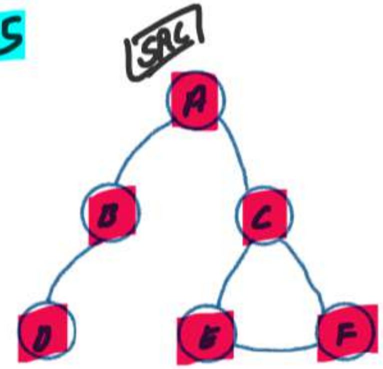
STEP3 Ignore B

Visited

Key	Value
A	T
B	~~F~~ T
C	~~F~~ T
D	~~F~~ T
E	~~F~~ T
F	~~F~~ T

use to track the visited child

Iterations



Queue

q	~~E~~	F	

FRONT Element

Cout
A B C D E

STEP1 front = E
q.pop()

STEP2 {C,F}

STEP3 Ignored C,F

Adjlist

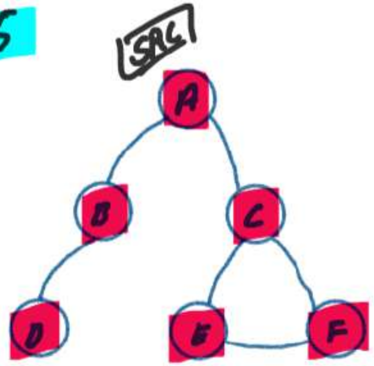
Key	Value
A	{B,C}
B	{D}
C	{E,F}
D	{B}
E	{C,F}
F	{C,E}

Visited

Key	Value
A	T
B	~~F~~ T
C	~~F~~ T
D	~~F~~ T
E	~~F~~ T
F	~~F~~ T

use to track the visited child

Iteration6



Queue

q	~~F~~	

FRONT Element

Cout

A B C D E F

AdjList

Key	Value
A	{ B, C }
B	{ D }
C	{ E, F }
D	{ B }
E	{ C, F }
F	{ C, E }

Visited

Key	Value
A	T
B	~~F~~ T
C	~~F~~ T
D	~~F~~ T
E	~~F~~ T
F	~~F~~ T

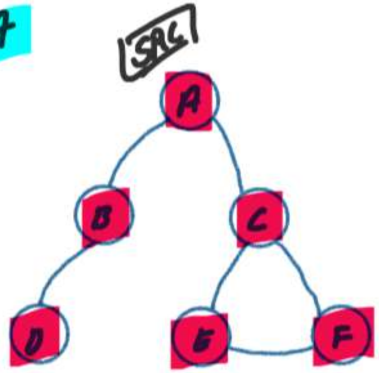
use to track the visited child

STEP1 front = F
q.pop()

STEP2 { C, E }

STEP3 Ignored C, E

Iteration7



Queue

q Empty

STOP

Queue is Empty Now

Cout

A B C D E F

Final output

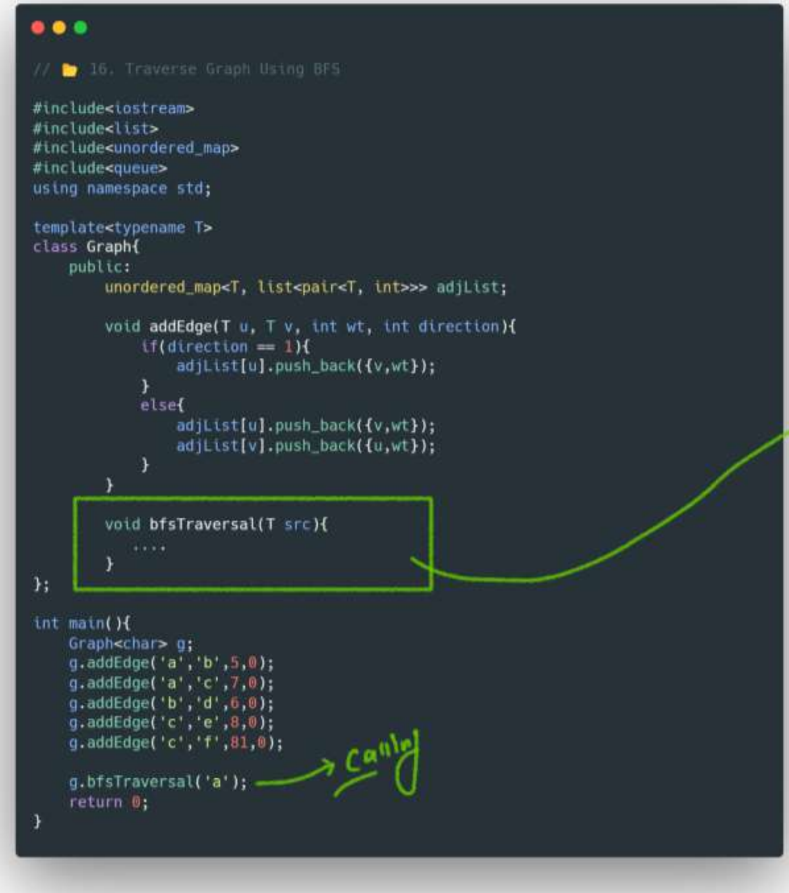
Adjlist

Key	Value
A	{B,C}
B	{ D}
C	{ E,F}
D	{B}
E	{C, F}
F	{C, E}

Visited

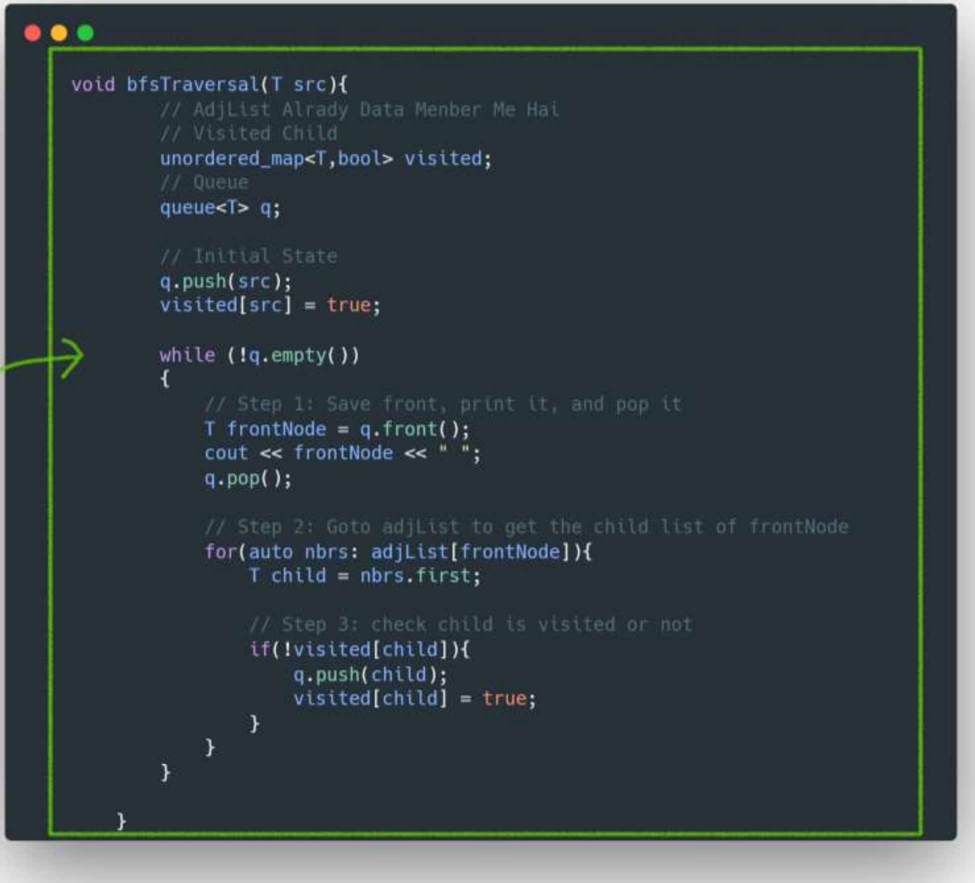
Key	Value
A	T
B	~~F~~ T
C	~~F~~ T
D	~~F~~ T
E	~~F~~ T
F	~~F~~ T

use to track the visited child



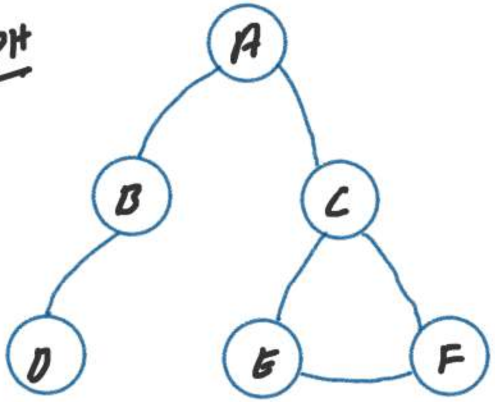
```
// 📁 16. Traverse Graph Using BFS

#include<iostream>
#include<list>
#include<unordered_map>
#include<queue>
using namespace std;

template<typename T>
class Graph{
    public:
        unordered_map<T, list<pair<T, int>>> adjList;

        void addEdge(T u, T v, int wt, int direction){
            if(direction == 1){
                adjList[u].push_back({v,wt});
            }
            else{
                adjList[u].push_back({v,wt});
                adjList[v].push_back({u,wt});
            }
        }

        void bfsTraversal(T src){
           ....
        }
};

int main(){
    Graph<char> g;
    g.addEdge('a','b',5,0);
    g.addEdge('a','c',7,0);
    g.addEdge('b','d',6,0);
    g.addEdge('c','e',8,0);
    g.addEdge('c','f',81,0);

    g.bfsTraversal('a');
    return 0;
}
```

Calling



```
void bfsTraversal(T src){
        // AdjList Alrady Data Menber Me Hai
        // Visited Child
        unordered_map<T,bool> visited;
        // Queue
        queue<T> q;

        // Initial State
        q.push(src);
        visited[src] = true;

        while (!q.empty())
        {
            // Step 1: Save front, print it, and pop it
            T frontNode = q.front();
            cout << frontNode << " ";
            q.pop();

            // Step 2: Goto adjList to get the child list of frontNode
            for(auto nbrs: adjList[frontNode]){
                T child = nbrs.first;

                // Step 3: check child is visited or not
                if(!visited[child]){
                    q.push(child);
                    visited[child] = true;
                }
            }
        }

    }
```

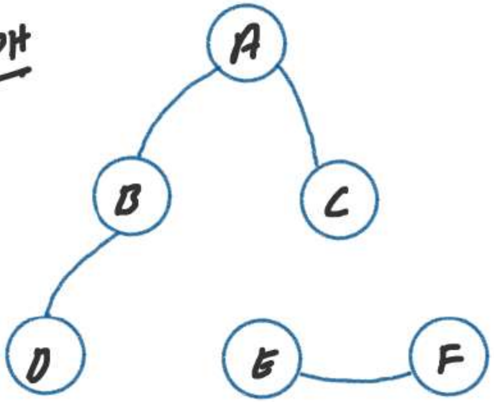
CONNECTED GRAPH



Expected output

A B C D E F

DISCONNECTED GRAPH



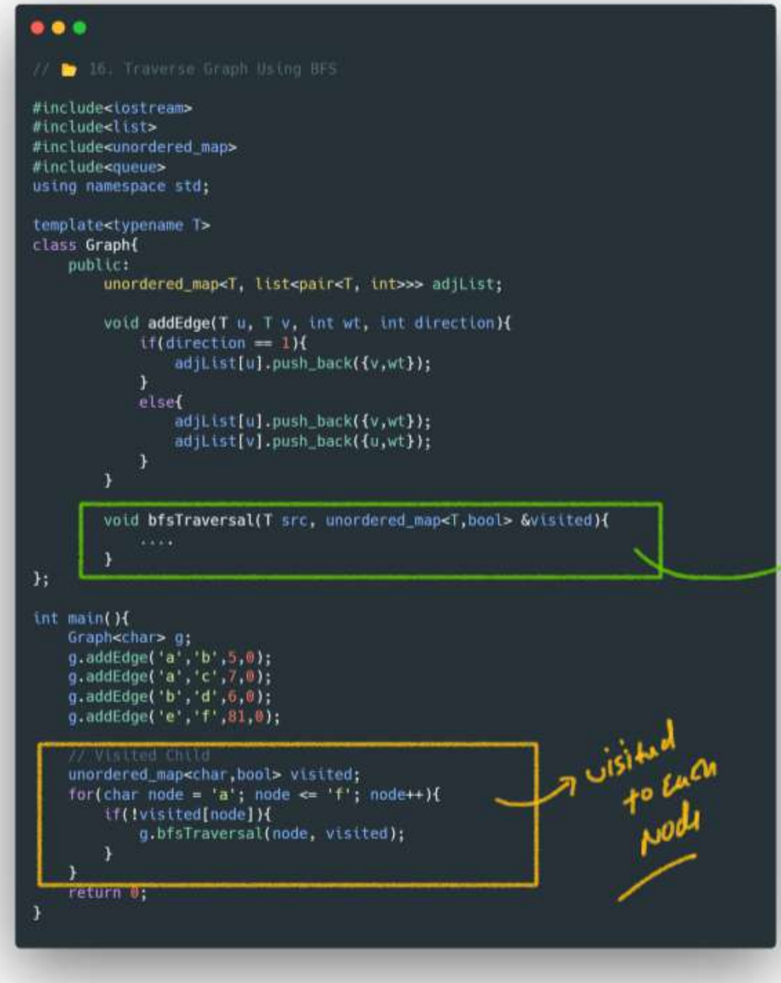
solve this issue into BFS

Expected output

A B C D E F

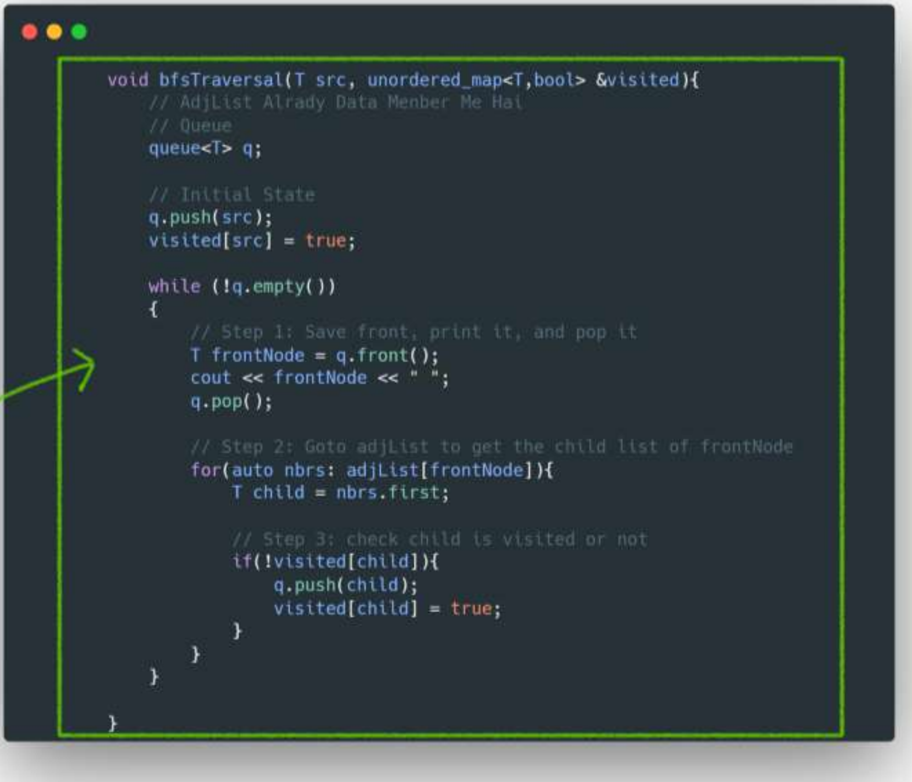
WRONG output

A B C D



```cpp
// 📂 16. Traverse Graph Using BFS

#include<iostream>
#include<list>
#include<unordered_map>
#include<queue>
using namespace std;

template<typename T>
class Graph{
    public:
        unordered_map<T, list<pair<T, int>>> adjList;

        void addEdge(T u, T v, int wt, int direction){
            if(direction == 1){
                adjList[u].push_back({v,wt});
            }
            else{
                adjList[u].push_back({v,wt});
                adjList[v].push_back({u,wt});
            }
        }

        void bfsTraversal(T src, unordered_map<T,bool> &visited){
            ....
        }
};

int main(){
    Graph<char> g;
    g.addEdge('a','b',5,0);
    g.addEdge('a','c',7,0);
    g.addEdge('b','d',6,0);
    g.addEdge('e','f',81,0);

    // Visited Child
    unordered_map<char,bool> visited;
    for(char node = 'a'; node <= 'f'; node++){
        if(!visited[node]){
            g.bfsTraversal(node, visited);
        }
    }
    return 0;
}
```

visited to each node

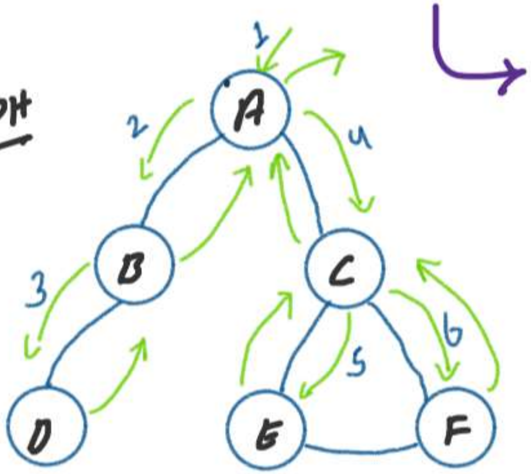


```cpp
    void bfsTraversal(T src, unordered_map<T,bool> &visited){
        // AdjList Alrady Data Menber Me Hai
        // Queue
        queue<T> q;

        // Initial State
        q.push(src);
        visited[src] = true;

        while (!q.empty())
        {
            // Step 1: Save front, print it, and pop it
            T frontNode = q.front();
            cout << frontNode << " ";
            q.pop();

            // Step 2: Goto adjList to get the child list of frontNode
            for(auto nbrs: adjList[frontNode]){
                T child = nbrs.first;

                // Step 3: check child is visited or not
                if(!visited[child]){
                    q.push(child);
                    visited[child] = true;
                }
            }
        }

    }
```

17. Traverse Graph Using DFS

→ using Recursion

GRAPH



A's child ⇒ B, C

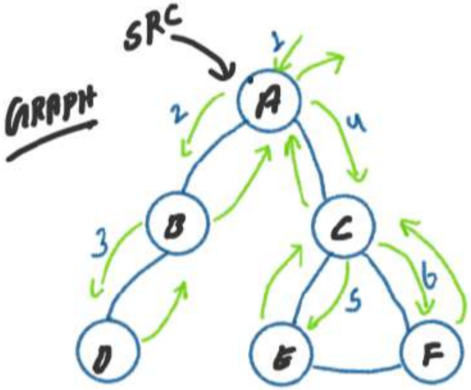
B's child ⇒ D

C's child ⇒ E, F

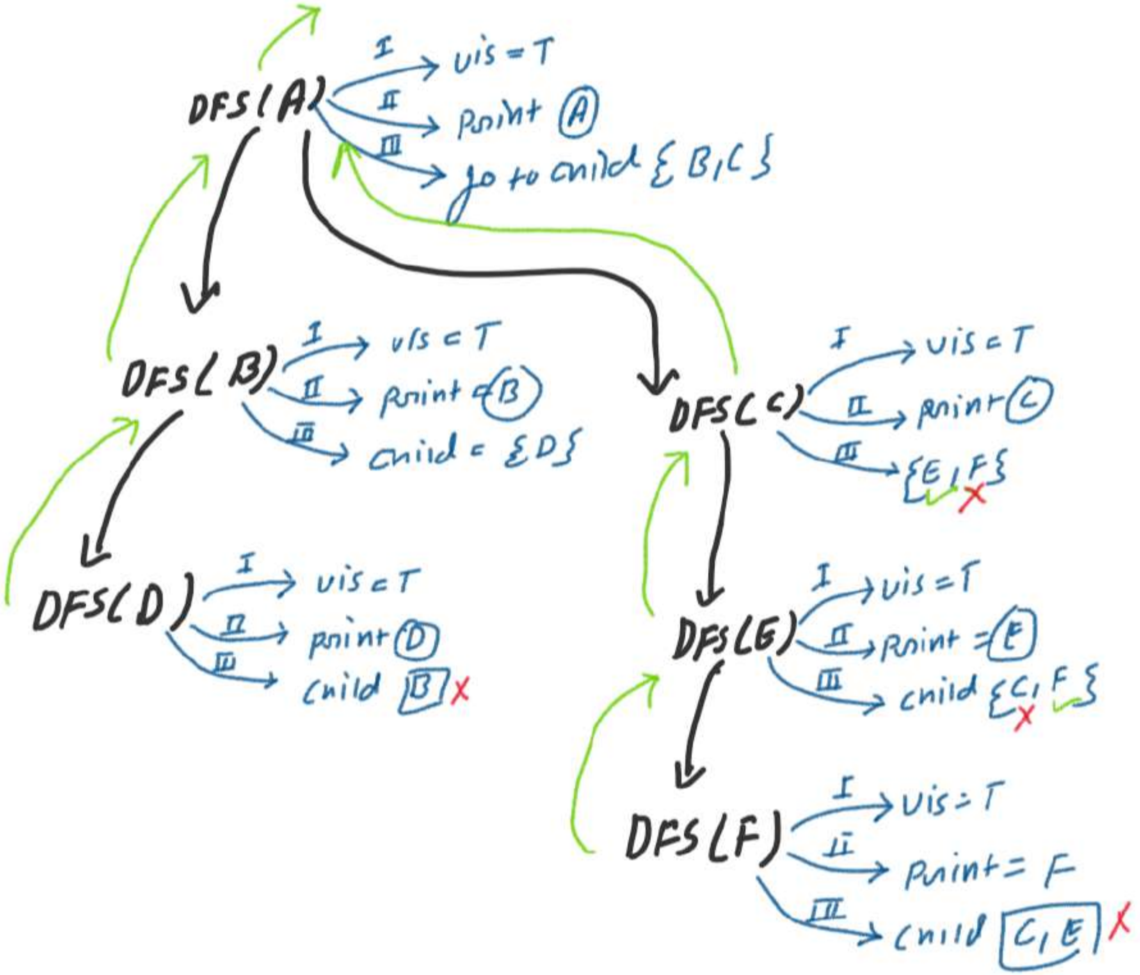
Outpot

↳ A B D C E F

Logic Build



DRY RUN

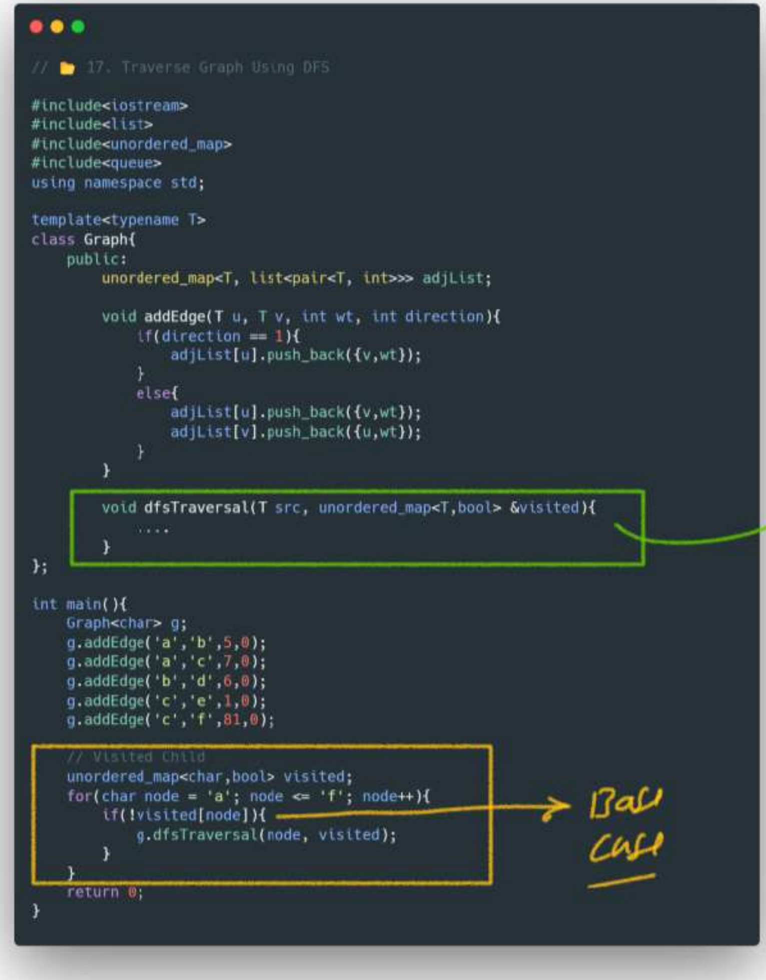


AdjList

Key	Value
A	{ B, C }
B	{ D }
C	{ E, F }
D	{ B }
E	{ C, F }
F	{ C, E }

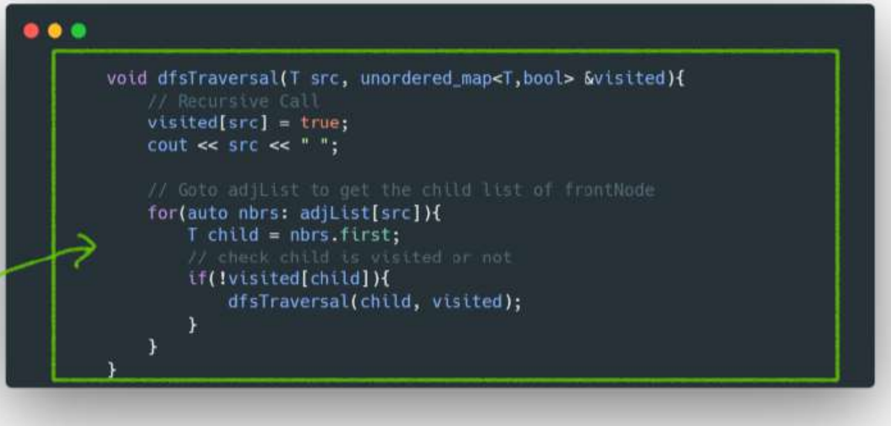
Visited

Key	Value
A	~~F~~ T
B	~~F~~ T
C	~~F~~ T
D	~~F~~ T
E	~~F~~ T
F	~~F~~ T



```cpp
// 📁 17. Traverse Graph Using DFS

#include<iostream>
#include<list>
#include<unordered_map>
#include<queue>
using namespace std;

template<typename T>
class Graph{
    public:
        unordered_map<T, list<pair<T, int>>> adjList;

        void addEdge(T u, T v, int wt, int direction){
            if(direction == 1){
                adjList[u].push_back({v,wt});
            }
            else{
                adjList[u].push_back({v,wt});
                adjList[v].push_back({u,wt});
            }
        }

        void dfsTraversal(T src, unordered_map<T,bool> &visited){
            ....
        }
};

int main(){
    Graph<char> g;
    g.addEdge('a','b',5,0);
    g.addEdge('a','c',7,0);
    g.addEdge('b','d',6,0);
    g.addEdge('c','e',1,0);
    g.addEdge('c','f',81,0);

    // Visited Child
    unordered_map<char,bool> visited;
    for(char node = 'a'; node <= 'f'; node++){
        if(!visited[node]){
            g.dfsTraversal(node, visited);
        }
    }
    return 0;
}
```

Base case



```cpp
        void dfsTraversal(T src, unordered_map<T,bool> &visited){
            // Recursive Call
            visited[src] = true;
            cout << src << " ";

            // Goto adjList to get the child list of frontNode
            for(auto nbrs: adjList[src]){
                T child = nbrs.first;
                // check child is visited or not
                if(!visited[child]){
                    dfsTraversal(child, visited);
                }
            }
        }
```