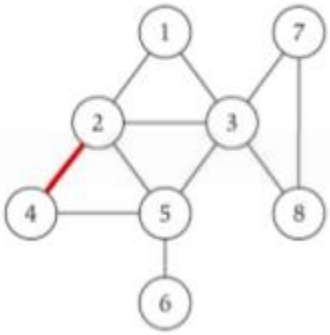
1. **Studi Kasus 1:**

Dengan menggunakan *undirected graph* dan *adjacency matrix* berikut, buatlah koding programmnya menggunakan bahasa C++.



	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

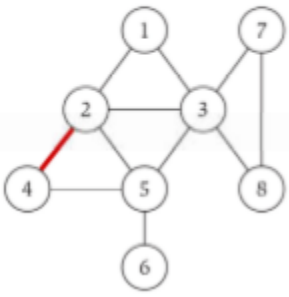
```
/*
NAMA    : SURIADI VAJRAKARNA
NPM     : 140810180038
KELAS   : B
TANGGAL : 5 APRIL 2020
STUDI KASUS 1 - PRAKTIKUM DESAIN DAN ANALISIS ALGORITMA
*/
#include "graph.hpp"
#include <iostream>

void print(int data) { std::cout << data << ' '; }

int main(int argc, char const **argv)
{
    const size_t graph_size = 8;
    Analgo::Graph<int> g(graph_size);

    g.add_edge(1, 2);
    g.add_edge(1, 3);
    g.add_edge(2, 3);
    g.add_edge(2, 4);
    g.add_edge(2, 5);
    g.add_edge(3, 5);
    g.add_edge(3, 7);
    g.add_edge(3, 8);
    g.add_edge(4, 5);
    g.add_edge(5, 6);
    g.add_edge(7, 8);

    try
    {
```

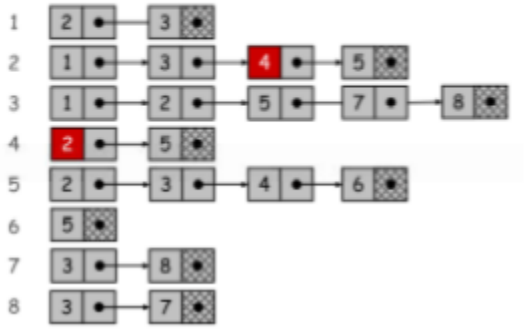
```
        std::cout << "Adjacency Matrix dari Graf tersebut:\n";
        for (const auto &node1 : g)
        {
            for (const auto &node2 : g)
            {
                std::cout << g.is_edge(node1.first, node2.first) << ' ';
            }
            std::cout << '\n';
        }
    }

    catch (const std::exception &e)
    {
        std::cerr << e.what() << '\n';
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

```
PS D:\MEGA\SEMESTER 4\ANALGO\Praktikum\AnalgoKu\AnalgoKu6> g++ -o adjmatrix adjmatrix.cpp
PS D:\MEGA\SEMESTER 4\ANALGO\Praktikum\AnalgoKu\AnalgoKu6> ./adjmatrix
Adjacency Matrix dari Graf tersebut:
0 1 1 0 0 0 0 0
1 0 1 1 1 0 0 0
1 1 0 0 1 0 1 1
0 1 0 0 1 0 0 0
0 1 1 1 0 1 0 0
0 0 0 0 1 0 0 0
0 0 1 0 0 0 0 1
0 0 1 0 0 0 1 0
```

2. Studi Kasus 2:

Dengan menggunakan *undirected graph* dan representasi *adjacency list*, buatlah koding programmnya menggunakan bahasa C++.





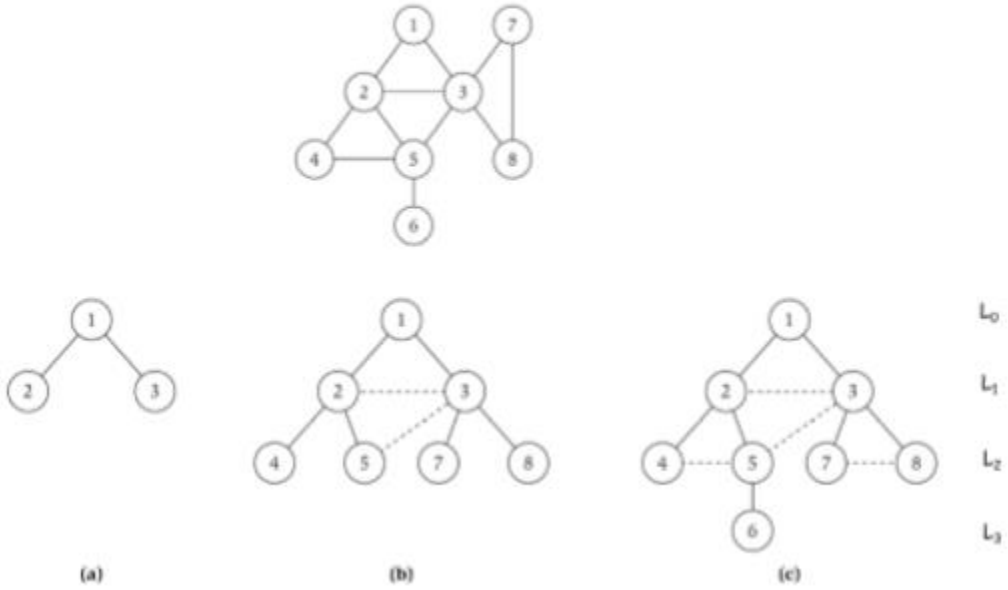

```
/*
NAMA    : SURIADI VAJRAKARNA
NPM     : 140810180038
KELAS   : B
TANGGAL : 5 APRIL 2020
STUDI KASUS 2 - PRAKTIKUM DESAIN DAN ANALISIS ALGORITMA
*/

#include "graph.hpp"
#include <iostream>

void print(int data) { std::cout << data << ' '; }

int main(int argc, char const **argv)
{
    const size_t graph_size = 8;
    Analgo::Graph<int> g(graph_size);

    g.add_edge(1, 2);
    g.add_edge(1, 3);
    g.add_edge(2, 3);
    g.add_edge(2, 4);
    g.add_edge(2, 5);
    g.add_edge(3, 5);
    g.add_edge(3, 7);
    g.add_edge(3, 8);
    g.add_edge(4, 5);
    g.add_edge(5, 6);
    g.add_edge(7, 8);
```

```
    try
    {
        std::cout << "Adjacency List dari Graf tersebut:\n";
        for (const auto &node1 : g)
        {
            std::cout << node1.first << "\t";
            for (const auto &node2 : g)
            {
                if (g.is_edge(node1.first, node2.first))
                {
                    std::cout << node2.first << " -> ";
                }
            }
            std::cout << '\n';
        }
    }

    catch (const std::exception &e)
    {
        std::cerr << e.what() << '\n';
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

```
PS D:\MEGA\SEMESTER 4\ANALGO\Praktikum\AnalgoKu\AnalgoKu6> g++ -o adjlist adjlist.cpp
PS D:\MEGA\SEMESTER 4\ANALGO\Praktikum\AnalgoKu\AnalgoKu6> ./adjlist
Adjacency List dari Graf tersebut:
1       2 -> 3 ->
2       1 -> 3 -> 4 -> 5 ->
3       1 -> 2 -> 5 -> 7 -> 8 ->
4       2 -> 5 ->
5       2 -> 3 -> 4 -> 6 ->
6       5 ->
7       3 -> 8 ->
8       3 -> 7 ->
```

3. Studi Kasus 3:

Buatlah program Breadth First Search dari algoritma BFS yang telah diberikan. Kemudian uji coba program Anda dengan menginputkan *undirected graph* sehingga menghasilkan tree BFS. Hitung dan berikan secara asimptotik berapa kompleksitas waktunya dalam Big-Θ!




```
/*
NAMA    : SURIADI VAJRAKARNA
NPM     : 140810180038
KELAS   : B
TANGGAL : 4 APRIL 2020
STUDI KASUS 3 - PRAKTIKUM DESAIN DAN ANALISIS ALGORITMA
*/

#include "graph.hpp"
#include <iostream>

void print(int data) { std::cout << data << ' '; }

int main(int argc, char const **argv)
{
    const size_t graph_size = 8;
    Analgo::Graph<int> g(graph_size);

    g.add_edge(1, 2);
    g.add_edge(1, 3);
    g.add_edge(2, 3);
    g.add_edge(2, 4);
    g.add_edge(2, 5);
    g.add_edge(3, 5);
```

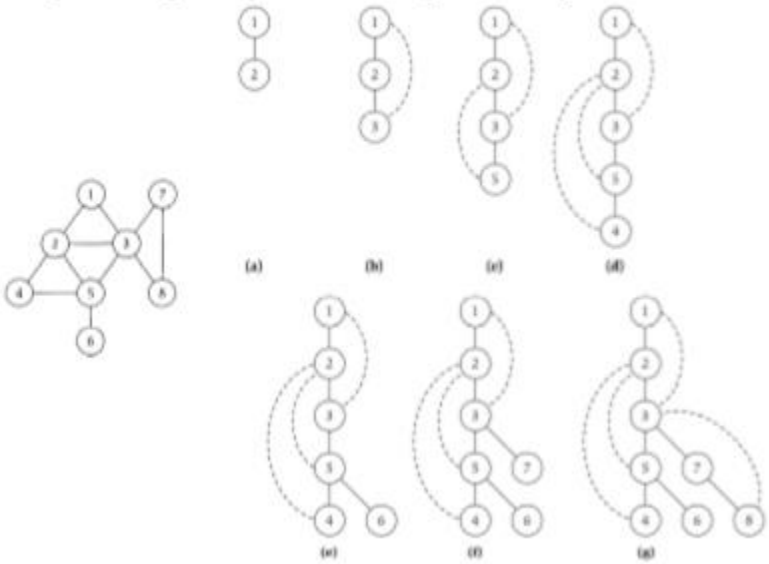
```
    g.add_edge(3, 7);
    g.add_edge(3, 8);
    g.add_edge(4, 5);
    g.add_edge(5, 6);
    g.add_edge(7, 8);

    try
    {
        std::cout << "Jalur Traversal Breadth First Search/BFS (dimulai dari simp
ul 1): ";
        g.bfs(1, [](const int &data) { std::cout << data << ' '; });
    }

    catch (const std::exception &e)
    {
        std::cerr << e.what() << '\n';
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

```
PS D:\MEGA\SEMESTER 4\ANALGO\Praktikum\AnalgoKu\AnalgoKu6> g++ -o bfs bfs.cpp
PS D:\MEGA\SEMESTER 4\ANALGO\Praktikum\AnalgoKu\AnalgoKu6> ./bfs
Jalur Traversal Breadth First Search/BFS (dimulai dari simpul 1): 1 2 3 4 5 7 8 6
```

Breadth First Search atau BFS merupakan metode pencarian secara melebar dengan mengunjungi node dari paling kiri ke paling kanan pada tingkat yang sama. Setelah itu, baru melanjutkan ke tingkat di bawahnya. Untuk memperkirakan *worst case*, BFS harus mempertimbangkan semua jalur yang mungkin untuk semua simpul sehingga nilai kompleksitas waktu dari BFS adalah **O(|V|+|E|)** di mana V adalah jumlah simpul dan E adalah jumlah ujung yang ada.

4. Studi Kasus 4:

Buatlah program Depth First Search dari algoritma DFS yang telah diberikan. Kemudian uji coba program Anda dengan menginputkan *undirected graph* sehingga menghasilkan tree DFS. Hitung dan berikan secara asimptotik berapa kompleksitas waktunya dalam Big-Θ!



```
/*
NAMA    : SURIADI VAJRAKARNA
NPM     : 140810180038
KELAS   : B
TANGGAL : 4 APRIL 2020
STUDI KASUS 4 - PRAKTIKUM DESAIN DAN ANALISIS ALGORITMA
*/

#include "graph.hpp"
#include <iostream>

void print(int data) { std::cout << data << ' '; }

int main(int argc, char const **argv)
{
    const size_t graph_size = 8;
    Analgo::Graph<int> g(graph_size);

    g.add_edge(1, 2);
    g.add_edge(1, 3);
    g.add_edge(2, 3);
    g.add_edge(2, 4);
    g.add_edge(2, 5);
    g.add_edge(3, 5);
    g.add_edge(3, 7);
```

```
    g.add_edge(3, 8);
    g.add_edge(4, 5);
    g.add_edge(5, 6);
    g.add_edge(7, 8);

    try
    {
        std::cout << "Jalur Traversal Depth First Search/DFS (dimulai dari simpul
 1): ";
        g.dfs(1, [](const int &data) { std::cout << data << ' '; });
    }

    catch (const std::exception &e)
    {
        std::cerr << e.what() << '\n';
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

```
PS D:\MEGA\SEMESTER 4\ANALGO\Praktikum\AnalgoKu\AnalgoKu6> g++ -o dfs dfs.cpp
PS D:\MEGA\SEMESTER 4\ANALGO\Praktikum\AnalgoKu\AnalgoKu6> ./dfs
Jalur Traversal Depth First Search/DFS (dimulai dari simpul 1): 1 3 8 7 5 6 4 2
```

Depth First Search atau DFS adalah metode pencarian mendalam dengan mengunjungi semua simpul dari paling kiri lalu ke bawah, baru melanjutkan ke kanannya hingga paling kanan. Kompleksitas algoritma DFS apabila dinotasikan dalam Big-O adalah O(|V|+|E|) di mana V adalah jumlah dari simpul dan E adalah jumlah dari ujung karena kita hanya memerlukan satu jalur tunggal dari akar sampai daun terakhir, ditambah dengan simpul-simpul saudara kandungnya yang belum ditelusuri.

5. **Library Graph (Tambahan): *graph.hpp***

```cpp
#pragma once

#include <algorithm>
#include <list>
#include <map>
#include <queue>
#include <stack>
#include <string>
#include <unordered_map>
#include <vector>

namespace Analgo
{

template <typename K, typename W, typename T>
class _base_graph
{
protected:
    _base_graph() {}

    _base_graph(size_t n);

    _base_graph(size_t n, const K &k);

    template <typename F>
    _base_graph(size_t n, F f);

public:
    void add_node(const K &k);

    void add_node(const K &k, const T &t);

    size_t order() { return node.size(); }

    size_t size() { return this->edge.size(); }

    T &operator[](const K &k);

    virtual bool is_edge(const K &k1, const K &k2) = 0;

    // graph iterator

    class iterator
```

```
{
    friend class _base_graph<K, W, T>;

public:
    iterator &operator++()
    {
        ++it;
        return *this;
    }
    iterator operator++(int)
    {
        iterator i = (*this);
        ++it;
        return i;
    }
    iterator &operator--()
    {
        --it;
        return *this;
    }
    iterator operator--(int)
    {
        iterator i = (*this);
        --it;
        return i;
    }

    std::pair<const K, T> &operator*() { return *it; }
    std::pair<const K, T> *operator->() { return &(*it); }

    bool operator==(const iterator &i) { return it == i.it; }
    bool operator!=(const iterator &i) { return it != i.it; }

private:
    typename std::map<K, T>::iterator begin, end;
    typename std::map<K, T>::iterator it;
};

virtual iterator begin();

virtual iterator end();

virtual iterator find(const K &k);

// edge
```

```
    class Edge
    {
    public:
        Edge(const K &k1, const K &k2, const W &w = 0) : ky1(k1), ky2(k2), wt(w)
{}

        bool operator==(const Edge &e) const
        {
            return key1() == e.key1() && key2() == e.key2();
        }
        bool operator!=(const Edge &e) const
        {
            return key1() != e.key1() || key2() != e.key2();
        }

        const K &key1() const { return ky1; }

        const K &key2() const { return ky2; }

        const K &key(const K &k);

        K &weight() { return wt; }

    private:
        K ky1;
        K ky2;
        W wt;
    };

    class edge_iterator
    {
        friend class _base_graph<K, W, T>;

    public:
        edge_iterator &operator++();
        edge_iterator operator++(int);
        edge_iterator &operator--();
        edge_iterator operator--(int);

        Edge &operator*() { return *it; }
        Edge *operator->() { return &(*it); }

        bool operator==(const edge_iterator &i) { return it == i.it; }
        bool operator!=(const edge_iterator &i) { return it != i.it; }
```

```
    private:
        typename std::list<Edge>::iterator begin, end;
        typename std::list<Edge>::iterator it;

        K key;
    };

    edge_iterator begin(const K &k);

    edge_iterator end(const K &k);

    // traverse algorithm

    template <typename F>
    F bfs(const K &k, F f);

    template <typename F>
    F dfs(const K &k, F f);

protected:
    std::map<K, T> node;
    std::list<Edge> edge;
};

template <typename K, typename W, typename T>
_base_graph<K, W, T>::_base_graph(size_t n)
{
    K key(1);

    for (size_t i = 0; i < n; i++)
        node[key++] = T();
}

template <typename K, typename W, typename T>
_base_graph<K, W, T>::_base_graph(size_t n, const K &k)
{
    K key(k);

    for (size_t i = 0; i < n; i++)
        node[key++] = T();
}

template <typename K, typename W, typename T>
template <typename F>
```

```
_base_graph<K, W, T>::_base_graph(size_t n, F f)
{
    for (size_t i = 0; i < n; i++)
        node[f()] = T();
}

template <typename K, typename W, typename T>
void _base_graph<K, W, T>::add_node(const K &k)
{
    if (node.find(k) != node.end())
        throw std::string("_base_graph::add_node - node already exists");
    node[k] = T();
}

template <typename K, typename W, typename T>
void _base_graph<K, W, T>::add_node(const K &k, const T &t)
{
    if (node.find(k) != node.end())
        throw std::string("_base_graph::add_node - node already exists");
    node[k] = t;
}

template <typename K, typename W, typename T>
typename _base_graph<K, W, T>::iterator _base_graph<K, W, T>::begin()
{
    iterator i;
    i.begin = node.begin();
    i.end = node.end();

    i.it = i.begin;

    return i;
}

template <typename K, typename W, typename T>
typename _base_graph<K, W, T>::edge_iterator
_base_graph<K, W, T>::begin(const K &k)
{
    edge_iterator i;
    i.begin = this->edge.begin();
    i.end = this->edge.end();
    i.key = k;

    i.it = i.begin;
    while (i.it != i.end && i.it->key1() != i.key && i.it->key2() != i.key)
```

```
    {
        i.it++;
    }

    return i;
}

template <typename K, typename W, typename T>
typename _base_graph<K, W, T>::iterator _base_graph<K, W, T>::end()
{
    iterator i;
    i.begin = node.begin();
    i.end = node.end();

    i.it = i.end;

    return i;
}

template <typename K, typename W, typename T>
typename _base_graph<K, W, T>::edge_iterator
_base_graph<K, W, T>::end(const K &k)
{
    edge_iterator i;
    i.begin = this->edge.begin();
    i.end = this->edge.end();
    i.key = k;

    i.it = i.end;

    return i;
}

template <typename K, typename W, typename T>
typename _base_graph<K, W, T>::iterator _base_graph<K, W, T>::find(const K &k)
{
    iterator i;
    i.begin = node.begin();
    i.end = node.end();
    i.it = node.find(k);

    return i;
}

template <typename K, typename W, typename T>
```

```
template <typename F>
F _base_graph<K, W, T>::bfs(const K &k, F f)
{
    K current;
    std::queue<K> q;
    std::unordered_map<K, bool> visited;

    q.push(k);
    visited[k] = true;

    while (!q.empty())
    {

        current = q.front();
        q.pop();
        f(current);

        for (iterator i = begin(); i != end(); i++)
            if (is_edge(current, i->first))
                if (!visited[i->first])
                {
                    q.push(i->first);
                    visited[i->first] = true;
                }
    }

    return f;
}

template <typename K, typename W, typename T>
template <typename F>
F _base_graph<K, W, T>::dfs(const K &k, F f)
{
    K current = k;
    std::unordered_map<K, bool> visited;
    std::stack<K> s;

    s.push(current);

    while (!s.empty())
    {
        current = s.top();
        s.pop();
        if (!visited[current])
        {
```

```
            f(current);
            visited[current] = true;
            for (iterator i = begin(); i != end(); ++i)
            {
                if (is_edge(current, i->first))
                {
                    s.push(i->first);
                }
            }
        }
    }

    return f;
}

template <typename K, typename W, typename T>
T &_base_graph<K, W, T>::operator[](const K &k)
{
    if (node.find(k) == node.end())
    {
        throw std::string("_base_graph::operator[] - node does not exist");
    }
    return node[k];
}

template <typename K, typename W, typename T>
const K &_base_graph<K, W, T>::Edge::key(const K &k)
{
    if (k != key1() && k != key2())
    {
        throw std::string("Graph::Edge::key - key supplied is invalid");
    }
    return k == key1() ? key2() : key1();
}

template <typename K, typename W, typename T>
typename _base_graph<K, W, T>::edge_iterator &
_base_graph<K, W, T>::edge_iterator::operator++()
{
    ++it;
    while (it != end && it->key1() != key && it->key2() != key)
    {
        ++it;
    }
```

```
    return *this;
}

template <typename K, typename W, typename T>
typename _base_graph<K, W, T>::edge_iterator
_base_graph<K, W, T>::edge_iterator::operator++(int)
{
    edge_iterator tmp = *this;

    ++it;
    while (it != end && it->key1() != key && it->key2() != key)
    {
        ++it;
    }

    return tmp;
}

template <typename K, typename W, typename T>
typename _base_graph<K, W, T>::edge_iterator &
_base_graph<K, W, T>::edge_iterator::operator--()
{
    --it;
    while (it != begin && it->key1() != key && it->key2() != key)
    {
        --it;
    }
    if (it == begin)
    {
        while (it != end && it->key1() != key && it->key2() != key)
        {
            ++it;
        }
    }

    return *this;
}

template <typename K, typename W, typename T>
typename _base_graph<K, W, T>::edge_iterator
_base_graph<K, W, T>::edge_iterator::operator--(int)
{
    edge_iterator tmp = *this;

    --it;
```

```
    while (it != begin && it->key1() != key && it->key2() != key)
    {
        --it;
    }
    if (it == begin)
    {
        while (it != end && it->key1() != key && it->key2() != key)
        {
            ++it;
        }
    }

    return tmp;
}

// undirected unweighted graph

template <typename K, typename T = void *>
class Graph : public _base_graph<K, void *, T>
{

    using Edge = typename _base_graph<K, void *, T>::Edge;

public:
    Graph() : _base_graph<K, void *, T>() {}

    Graph(size_t n) : _base_graph<K, void *, T>(n) {}

    Graph(size_t n, const K &k) : _base_graph<K, void *, T>(n, k) {}

    template <typename F>
    Graph(size_t n, F f) : _base_graph<K, void *, T>(n, f) {}

    void add_edge(const K &k1, const K &k2);

    bool is_edge(const K &k1, const K &k2);

    void remove_edge(const K &k1, const K &k2);
};

template <typename K, typename T>
void Graph<K, T>::add_edge(const K &k1, const K &k2)
{
    if (this->node.find(k1) == this->node.end() ||
        this->node.find(k2) == this->node.end())
```

```
    {
        throw std::string("add_edge: this->node does not exist");
    }

    // Store lower key value in first key of edge
    if (k1 < k2)
        this->edge.push_back(Edge(k1, k2));
    else
        this->edge.push_back(Edge(k2, k1));
}

template <typename K, typename T>
bool Graph<K, T>::is_edge(const K &k1, const K &k2)
{
    if (this->node.find(k1) == this->node.end() ||
        this->node.find(k2) == this->node.end())
    {
        throw std::string("is_edge: this->node does not exist");
    }

    if (k1 < k2)
        return std::find(this->edge.begin(), this->edge.end(), Edge(k1, k2)) !=
               this->edge.end();
    return std::find(this->edge.begin(), this->edge.end(), Edge(k2, k1)) !=
           this->edge.end();
}

template <typename K, typename T>
void Graph<K, T>::remove_edge(const K &k1, const K &k2)
{
    if (this->node.find(k1) == this->node.end() ||
        this->node.find(k2) == this->node.end())
        throw std::string("add_edge: this->node does not exist");

    if (k1 < k2)
        this->edge.remove(Edge(k1, k2));
    else
        this->edge.remove(Edge(k2, k1));
}

// weighted graph

template <typename K, typename W, typename T = void *>
class WeightedGraph : public _base_graph<K, W, T>
{
```

```
    using Edge = typename _base_graph<K, W, T>::Edge;

public:
    WeightedGraph() : _base_graph<K, W, T>() {}

    WeightedGraph(size_t n) : _base_graph<K, W, T>(n) {}

    WeightedGraph(size_t n, const K &k) : _base_graph<K, W, T>(n, k) {}

    template <typename F>
    WeightedGraph(size_t n, F f) : _base_graph<K, W, T>(n, f) {}

    void add_edge(const K &k1, const K &k2, const W &w);

    bool is_edge(const K &k1, const K &k2);

    void remove_edge(const K &k1, const K &k2);

    W &weight(const K &k1, const K &k2);
};

template <typename K, typename W, typename T>
void WeightedGraph<K, W, T>::add_edge(const K &k1, const K &k2, const W &w)
{
    if (this->node.find(k1) == this->node.end() ||
        this->node.find(k2) == this->node.end())
        throw std::string("add_edge: this->node does not exist");

    // Store lower key value in first key of edge
    if (k1 < k2)
        this->edge.push_back(edge(k1, k2, w));
    else
        this->edge.push_back(edge(k2, k1, w));
}

template <typename K, typename W, typename T>
bool WeightedGraph<K, W, T>::is_edge(const K &k1, const K &k2)
{
    if (this->node.find(k1) == this->node.end() ||
        this->node.find(k2) == this->node.end())
        throw std::string("is_edge: this->node does not exist");

    if (k1 < k2)
        return std::find(this->edge.begin(), this->edge.end(), edge(k1, k2)) !=
```

```
                this->edge.end();
    return std::find(this->edge.begin(), this->edge.end(), edge(k2, k1)) !=
           this->edge.end();
}

template <typename K, typename W, typename T>
void WeightedGraph<K, W, T>::remove_edge(const K &k1, const K &k2)
{
    if (this->node.find(k1) == this->node.end() ||
        this->node.find(k2) == this->node.end())
        throw std::string("add_edge: this->node does not exist");

    if (k1 < k2)
        this->edge.remove(Edge(k1, k2));
    else
        this->edge.remove(Edge(k2, k1));
}

template <typename K, typename W, typename T>
W &WeightedGraph<K, W, T>::weight(const K &k1, const K &k2)
{
    if (this->node.find(k1) == this->node.end() ||
        this->node.find(k2) == this->node.end())
        throw std::string("weight: this->node does not exist");
    if (!is_edge(k1, k2))
        throw std::string("weight: Edge does not exist");

    if (k1 < k2)
        return std::find(this->edge.begin(), this->edge.end(), Edge(k1, k2))
            ->weight();
    return std::find(this->edge.begin(), this->edge.end(), Edge(k2, k1))
        ->weight();
}

// directed graph

template <typename K, typename T = void *>
class DirectedGraph : public _base_graph<K, void *, T>
{

    using Edge = typename _base_graph<K, void *, T>::Edge;

public:
    DirectedGraph() : _base_graph<K, void *, T>() {}
```

```
    DirectedGraph(size_t n) : _base_graph<K, void *, T>(n) {}

    DirectedGraph(size_t n, const K &k) : _base_graph<K, void *, T>(n, k) {}

    template <typename F>
    DirectedGraph(size_t n, F f) : _base_graph<K, void *, T>(n, f) {}

    void add_edge(const K &k1, const K &k2);

    bool is_edge(const K &k1, const K &k2);

    void remove_edge(const K &k1, const K &k2);
};

template <typename K, typename T>
void DirectedGraph<K, T>::add_edge(const K &k1, const K &k2)
{
    if (this->node.find(k1) == this->node.end() ||
        this->node.find(k2) == this->node.end())
        throw std::string("add_edge: this->node does not exist");

    this->edge.push_back(edge(k1, k2));
}

template <typename K, typename T>
bool DirectedGraph<K, T>::is_edge(const K &k1, const K &k2)
{
    if (this->node.find(k1) == this->node.end() ||
        this->node.find(k2) == this->node.end())
        throw std::string("is_edge: this->node does not exist");

    return std::find(this->edge.begin(), this->edge.end(), Edge(k1, k2)) !=
           this->edge.End();
}

template <typename K, typename T>
void DirectedGraph<K, T>::remove_edge(const K &k1, const K &k2)
{
    if (this->node.find(k1) == this->node.end() ||
        this->node.find(k2) == this->node.end())
        throw std::string("add_edge: this->node does not exist");

    this->edge.remove(edge(k1, k2));
}
```

```
template <typename K, typename W, typename T = void *>
class WeightedDirectedGraph : public _base_graph<K, W, T>
{

    using Edge = typename _base_graph<K, void *, T>::Edge;

public:
    WeightedDirectedGraph() : _base_graph<K, W, T>() {}

    WeightedDirectedGraph(size_t n) : _base_graph<K, W, T>(n) {}

    WeightedDirectedGraph(size_t n, const K &k) : _base_graph<K, W, T>(n, k) {}

    template <typename F>
    WeightedDirectedGraph(size_t n, F f) : _base_graph<K, W, T>(n, f) {}

    void add_edge(const K &k1, const K &k2, const W &w);

    bool is_edge(const K &k1, const K &k2);

    void remove_edge(const K &k1, const K &k2);

    W &weight(const K &k1, const K &k2);
};

template <typename K, typename W, typename T>
void WeightedDirectedGraph<K, W, T>::add_edge(const K &k1, const K &k2,
                                              const W &w)
{
    if (this->node.find(k1) == this->node.end() ||
        this->node.find(k2) == this->node.end())
        throw std::string("add_edge: this->node does not exist");

    this->edge.push_back(edge(k1, k2, w));
}

template <typename K, typename W, typename T>
bool WeightedDirectedGraph<K, W, T>::is_edge(const K &k1, const K &k2)
{
    if (this->node.find(k1) == this->node.end() ||
        this->node.find(k2) == this->node.end())
        throw std::string("is_edge: this->node does not exist");

    return std::find(this->edge.begin(), this->edge.end(), Edge(k1, k2)) !=
           this->edge.end();
```

```
}

template <typename K, typename W, typename T>
void WeightedDirectedGraph<K, W, T>::remove_edge(const K &k1, const K &k2)
{
    if (this->node.find(k1) == this->node.end() ||
        this->node.find(k2) == this->node.end())
        throw std::string("add_edge: this->node does not exist");

    this->edge.remove(Edge(k1, k2));
}

template <typename K, typename W, typename T>
W &WeightedDirectedGraph<K, W, T>::weight(const K &k1, const K &k2)
{
    if (this->node.find(k1) == this->node.end() ||
        this->node.find(k2) == this->node.end())
        throw std::string("weight: Node does not exist");
    if (!is_edge(k1, k2))
        throw std::string("weight: Edge does not exist");

    return std::find(this->edge.begin(), this->edge.end(), Edge(k1, k2))
        ->weight();
}

}
```