

Documentation of Graph Explorer

Table of Contents

Documentation of Graph Explorer	0
Table of Contents	1
graph	6
Overview	6
Constructors	6
Parameters	7
Example	7
Output	7
Complexity	7
Destructor	7
Complexity	8
push_back	8
Parameters	8
Return Value	8
Example	8
Output	9
Complexity	9
add_edge	9
Parameters	9
Return Value	10
None	10
Exception Safety	10
Example	10
Output	10
Complexity	11
erase_edge	11
Parameters	11
Return Value	11
None	11
Exception Safety	11
Example	11
Output	11
Complexity	12
add_undirected_edge	12
Parameters	12

Return Value	13
Exception Safety	13
Example	13
Output	13
Complexity	14
erase_undirected_edge	14
Parameters	14
Return Value	14
Exception Safety	14
Example	14
Output	14
Complexity	15
nodes	15
Parameters	15
Return Value	15
Example	15
Output	16
Complexity	16
size	16
Parameters	16
Return Value	16
Example	16
Output	16
Complexity	17
operator[]	17
Parameters	18
Return Value	18
Exception Safety	18
Example	18
Output	18
Complexity	19
count	19
Parameters	19
Return Value	19
Example	19
Output	19
Complexity	20
get_index	20
Parameters	20

Return Value	20
Example	20
Output	21
Complexity	21
vertex	21
Methods	22
begin	22
Parameters	22
Return Value	22
Example	22
Output	22
Complexity	23
end	23
Parameters	23
Return Value	23
Example	23
Output	24
Complexity	25
operator[]	25
Parameters	25
Return Value	25
Exception Safety	25
Example	25
Output	25
Complexity	26
operator =	26
Parameters	26
Return Value	26
None	27
Exception Safety	27
Example	27
Output	27
Complexity	28
unordered_graph	29
Overview	29
Constructors	29
Parameters	29
Example	30
Output	30

Complexity	30
Destructor	30
Complexity	30
push_back	31
Parameters	31
Return Value	31
Example	31
Output	31
Complexity	32
add_edge	32
Parameters	32
Return Value	32
None	32
Exception Safety	32
Example	32
Output	32
Complexity	33
erase_edge	33
Parameters	33
Return Value	33
None	33
Exception Safety	33
Example	33
Output	34
Complexity	34
add_undirected_edge	34
Parameters	35
Return Value	35
Exception Safety	35
Example	35
Output	35
Complexity	36
erase_undirected_edge	36
Parameters	36
Return Value	36
Exception Safety	36
Example	36
Output	36
Complexity	37

nodes	37
Parameters	37
Return Value	37
Example	37
Output	38
Complexity	38
size	38
Parameters	38
Return Value	38
Example	38
Output	38
Complexity	39
operator[]	39
Parameters	39
Return Value	39
Exception Safety	39
Example	39
Output	39
Complexity	40

graph

```
template <EqualityComparable T, NumericType Edge> class graph
```

```
template <LessThanComparable T, NumericType Edge> class graph<T, Edge>
```

Overview

graph is a container that stores a set of vertices and edges between them. the types of vertices and edges are templated so any compatible type be used.

The definition of compatibility is the following:

- the value type of a vertex should support “==” operator between 2 instances, this is used to eliminate duplicates. If the value type also implements “<”, then the library builds a lookup table to efficiently eliminate duplicates.
- Edge type should be numeric, this means that it should support addition, multiplication, division, and subtraction. This is used by graph algorithm library to implement various algorithm on top of the representation.

Constructors

<i>default(1):</i>	graph()
<i>initializer_list(2):</i>	graph(const initializer_list &inp);
<i>copy(3):</i>	graph(const graph &g)
<i>move(4):</i>	graph(graph &&g)

(1) Empty graph constructor

- Constructs an empty graph with no vertex and no edges

(2) Initializer list constructor

- Constructs a graph with copy of elements in the initializer list in the same order while eliminating duplicates.

(3) Copy graph constructor

- Constructs a graph with copy of each vertex and edge in graph g

(4) Move graph constructor

- Constructor that acquires the vertices and edges of g. No elements are constructed, their ownership is directly transferred.

Parameters

inp	list of vertex values
g	another graph object of same type

Example

```
#include "graph_matrix.h"
#include <iostream>
using namespace graphmatrix;

int main(){
    graph<int, int> empty_graph;
    graph<int, int> g2{1,2,3};
    auto g4 = g2;
    auto g5 = move(g1);
    std::cout << g1[5].val << '\n';
    return 0;
}
```

Output

12

Complexity

Constant for default(1) and move(4) constructors. $O(n \log n)$ where n is the size of nodes for initializer list(2) constructor if T supports both equality and less than operator. $O(n^2)$ where n is the size of nodes for initializer list(2) constructor if T does not support less than operator. Linear in number of vertices and edge for copy(3) constructor.

Destructor

~graph()

This calls `allocator_traits::destroy` on each of the vertices and edges, and deallocates all the storage used by the graph

Complexity

Linear in number of vertices + number of edges in the graph.

push_back

```
uint32_t push_back(const T &val)
```

Pushes the new vertex to the end of the graph if the vertex is not already present and returns the index of the inserted vertex. If the vertex is already present in the graph, then it just returns the index of the new vertex inserted.

Parameters

val	value of the vertex to be inserted
-----	------------------------------------

Return Value

Index of the vertex that is either inserted newly or already present.

Example

```
#include "graph_matrix.h"
#include <iostream>
#include <cassert>
using namespace graphmatrix;

int main(){
    graph<string, double> g;
    g.reserve(3);
    cout << g.size() << '\n';

    g.push_back("New York");
    g.push_back("Seattle");
    g.push_back("Chicago");

    // size of the graph
    cout << g.size() << '\n';

    // get the vertex indices for the 2 cities.
```

```

auto src = g.get_index("New York");
auto target = g.get_index("Chicago");

if(src && target){
    cout << g[*target].val << '\n';
    g.add_edge(*src, *target, 22.55);
    cout << g[*src][*target] << '\n';
    g.erase_edge(*src, *target);
    cout << g[*src][*target] << '\n';
}else assert(false);
return 0;
}

```

Output

```

0
3
Chicago
22.55
0

```

Complexity

$O(\log n)$ amortized time and $O(n \log n)$ worst case time.

add_edge

```
void add_edge(uint32_t src, uint32_t target, const Edge &weight)
```

Adds a directed edge from vertex with index src to vertex with index target with edge weight as specified in the parameters

Parameters

src	index of source vertex
target	index of target index
weight	weight of the edge to be added/updated

Return Value

None

Exception Safety

If the index of vertex is out of range of the graph, then it throws `vertex_out_of_range_error()`.

Example

```
#include "graph_matrix.h"
#include <iostream>
#include <cassert>
using namespace graphmatrix;

int main(){
    graph<string, double> g;
    g.reserve(3);
    cout << g.size() << '\n';

    g.push_back("New York");
    g.push_back("Seattle");
    g.push_back("Chicago");

    // size of the graph
    cout << g.size() << '\n';

    // get the vertex indices for the 2 cities.
    auto src = g.get_index("New York");
    auto target = g.get_index("Chicago");

    if(src && target){
        cout << g[*target].val << '\n';
        g.add_edge(*src, *target, 22.55);
        cout << g[*src][*target] << '\n';
        g.erase_edge(*src, *target);
        cout << g[*src][*target] << '\n';
    }else assert(false);
    return 0;
}
```

Output

0
3

Chicago
22.55
0

Complexity

$O(1)$ amortized time complexity, $O(n^2)$ worst case time.

erase_edge

```
void erase_edge(uint32_t src, uint32_t target)
```

Removes the edge from vertex with index src to vertex with index target

Parameters

src	index of source vertex
target	index of target index

Return Value

None

Exception Safety

If the index of vertex is out of range of the graph, then it throws `vertex_out_of_range_error()`.

Example

```
#include "graph_matrix.h"
#include <iostream>
#include <cassert>
using namespace graphmatrix;

int main(){
    graph<string, double> g;
    g.reserve(3);
    cout << g.size() << '\n';

    g.push_back("New York");
    g.push_back("Seattle");
    g.push_back("Chicago");
```

```

// size of the graph
cout << g.size() << '\n';

// get the vertex indices for the 2 cities.
auto src = g.get_index("New York");
auto target = g.get_index("Chicago");

if(src && target){
    cout << g[*target].val << '\n';
    g.add_edge(*src, *target, 22.55);
    cout << g[*src][*target] << '\n';
    g.erase_edge(*src, *target);
    cout << g[*src][*target] << '\n';
}else assert(false);
return 0;
}

```

Output

```

0
3
Chicago
22.55
0

```

Complexity

$O(1)$ amortized time complexity, $O(n^2)$ worst case time.

add_undirected_edge

```
void add_undirected_edge(uint32_t src, uint32_t target, const Edge &weight)
```

Adds a undirected edge between vertex with index src and vertex with index target with edge weight as specified in the parameters

Parameters

src	index of source vertex
target	index of target index
weight	weight of the edge to be added/updated

Return Value

None

Exception Safety

If the index of vertex is out of range of the graph, then it throws `vertex_out_of_range_error()`.

Example

```
#include "graph_matrix.h"
#include <iostream>
#include <cassert>
using namespace graphmatrix;

int main(){
    graph<string, double> g;
    g.reserve(3);
    cout << g.size() << '\n';

    g.push_back("New York");
    g.push_back("Seattle");
    g.push_back("Chicago");

    // size of the graph
    cout << g.size() << '\n';

    // get the vertex indices for the 2 cities.
    auto src = g.get_index("New York");
    auto target = g.get_index("Chicago");

    if(src && target){
        cout << g[*target].val << '\n';
        g.add_undirected_edge(*src, *target, 22.55);
        cout << g[*src][*target] << '\n';
        g.erase_undirected_edge(*target, *src);
        cout << g[*src][*target] << '\n';
    }else assert(false);
    return 0;
}
```

Output

```
0
3
```

Chicago
22.55
0

Complexity

$O(1)$ amortized time complexity, $O(n^2)$ worst case time.

erase_undirected_edge

```
void erase_undirected_edge(uint32_t src, uint32_t target)
```

Removes the edge between vertex with index src and vertex with index target

Parameters

src	index of source vertex
target	index of target index

Return Value

None

Exception Safety

If the index of vertex is out of range of the graph, then it throws `vertex_out_of_range_error()`.

Example

```
#include "graph_matrix.h"
#include <iostream>
#include <cassert>
using namespace graphmatrix;

int main(){
    graph<string, double> g;
    g.reserve(3);
    cout << g.size() << '\n';

    g.push_back("New York");
    g.push_back("Seattle");
    g.push_back("Chicago");
```

```

// size of the graph
cout << g.size() << '\n';

// get the vertex indices for the 2 cities.
auto src = g.get_index("New York");
auto target = g.get_index("Chicago");

if(src && target){
    cout << g[*target].val << '\n';
    g.add_undirected_edge(*src, *target, 22.55);
    cout << g[*src][*target] << '\n';
    g.erase_undirected_edge(*target, *src);
    cout << g[*src][*target] << '\n';
}else assert(false);
return 0;
}

```

Output

```

0
3
Chicago
22.55
0

```

Complexity

$O(1)$ amortized time complexity, $O(n^2)$ worst case time.

nodes

```
const vector<T>& nodes() const
```

Gets reference to the vector containing values of the nodes. The reference is constant and hence it cannot be used to update the vertex values.

Parameters

None

Return Value

Reference to vector containing values of vertices of the graph

Example

```
#include "graph_matrix.h"
#include <iostream>
#include <cassert>
using namespace graphmatrix;

int main(){
    graph<string, double> g;
    g.push_back("New York");
    g.push_back("Seattle");
    g.push_back("Chicago");

    auto &nodes = g.nodes();
    for(auto& v : nodes) cout << v << ' ';
    cout << '\n';

    return 0;
}
```

Output

New York Seattle Chicago

Complexity

Constant time.

size

```
inline size_t size()
```

```
const inline size_t size() const
```

Gets the size of the graph which is equal to number of vertices in the graph.

Parameters

None

Return Value

Size of the graph

Example

```
#include "graph_matrix.h"
```

```

#include <iostream>
#include <cassert>
using namespace graphmatrix;

int main(){
    graph<string, double> g;
    g.reserve(3);
    cout << g.size() << '\n';

    g.push_back("New York");
    g.push_back("Seattle");
    g.push_back("Chicago");

    // size of the graph
    cout << g.size() << '\n';

    // get the vertex indices for the 2 cities.
    auto src = g.get_index("New York");
    auto target = g.get_index("Chicago");

    if(src && target){
        cout << g[*target].val << '\n';
        g.add_undirected_edge(*src, *target, 22.55);
        cout << g[*src][*target] << '\n';
        g.erase_undirected_edge(*target, *src);
        cout << g[*src][*target] << '\n';
    }else assert(false);
    return 0;
}

```

Output

```

0
3
Chicago
22.55
0

```

Complexity

Constant time.

operator[]

vertex<T, Edge> operator[](uint32_t i)

Gets the vertex object that contains information about the vertex like its value and edges originating from it. It can also be used to update the edge value using the subscript operator overloaded in the vertex class.

Parameters

i	index of the vertex to get information about
---	----------------------------------------------

Return Value

an object of class vertex<T, Edge>

Exception Safety

If the index of vertex is out of range of the graph, then it throws vertex_out_of_range_error().

Example

```
#include "graph_matrix.h"
#include <iostream>
#include <cassert>
using namespace graphmatrix;

int main(){
    graph<string, double> g;
    g.push_back("New York");
    g.push_back("Seattle");
    g.push_back("Chicago");

    // size of the graph
    cout << g.size() << '\n';
    vertex<string, double> v = g[0];
    cout << v.val << '\n';
    v[1] = 100;
    v[2] = 2.33;
    cout << g[0][1] << '\n';
    for(auto &p : g[0]){
        cout << g[0].val << "->" << g[p.first].val << ": " << p.second <<
'\n';
    }
    return 0;
}
```

```
}
```

Output

```
3
New York
100
New York->Chicago: 2.33
New York->Seattle: 100
```

Complexity

Constant time

count

```
uint8_t count(const T& val)
```

```
const uint8_t count(const T& val) const
```

Returns whether the count of given vertex value in the graph. Due to uniqueness constraint, the return value is either 0 or 1.

Parameters

val	value of the vertex to be searched
-----	------------------------------------

Return Value

0 if the vertex is not present, 1 if it is present.

Example

```
#include "graph_matrix.h"
#include <iostream>
#include <cassert>
using namespace graphmatrix;

int main(){
    graph<string, double> g;
    g.push_back("New York");
    g.push_back("Seattle");
    g.push_back("Chicago");
```

```

// size of the graph
cout << g.size() << '\n';
vertex<string, double> v = g[0];
cout << v.val << '\n';
if(g.count("Boston") == 0)
    cout << "Boston is not present" << '\n';
v[1] = 100;
auto idx = g.get_index("Chicago");
v[*idx] = 2.33;
cout << g[0][1] << '\n';
for(auto &p : g[0]){
    cout << g[0].val << "->" << g[p.first].val << ": " << p.second <<
'\n';
}
return 0;
}

```

Output

```

3
New York
Boston is not present
100
New York->Chicago: 2.33
New York->Seattle: 100

```

Complexity

$\log(n)$ time where n is the number of vertices in the graph

get_index

```
optional<uint32_t> get_index(const T& val)
```

Returns the index of the given vertex. If the vertex is not present then it returns nullopt.

Parameters

val	value of the vertex to be searched
-----	------------------------------------

Return Value

std::nullopt if the vertex is not present or pointer to index of the vertex.

Example

```

#include "graph_matrix.h"
#include <iostream>
#include <cassert>
using namespace graphmatrix;

int main(){
    graph<string, double> g;
    g.push_back("New York");
    g.push_back("Seattle");
    g.push_back("Chicago");

    // size of the graph
    cout << g.size() << '\n';
    vertex<string, double> v = g[0];
    cout << v.val << '\n';
    if(g.count("Boston") == 0)
        cout << "Boston is not present" << '\n';
    v[1] = 100;
    auto idx = g.get_index("Chicago");
    v[*idx] = 2.33;
    cout << g[0][1] << '\n';
    for(auto &p : g[0]){
        cout << g[0].val << "->" << g[p.first].val << ": " << p.second <<
'\n';
    }
    return 0;
}

```

Output

```

3
New York
Boston is not present
100
New York->Chicago: 2.33
New York->Seattle: 100

```

Complexity

$\log(n)$ time where n is the number of vertices in the graph

vertex

```
template <class T, NumericType Edge> class vertex
```

Stores the information about the vertex of the graph. This is the class returned by graph object when subscript operator is used on it. We recommend not using this class by constructing object of this class on your own as this is meant for usage with graph and will be maintained accordingly.

Methods

begin

```
typename unordered_map<uint32_t,Edge>::iterator begin()
```

Returns iterator to beginning of edges originating from the vertex which it represents.

Parameters

None

Return Value

Iterator to beginning hashmap of index of target vertex and the edge weight.

Example

```
#include "graph_matrix.h"
#include <iostream>
#include <cassert>
using namespace graphmatrix;

int main(){
    graph<string, double> g;
    g.push_back("New York");
    g.push_back("Seattle");
    g.push_back("Chicago");

    // size of the graph
    cout << g.size() << '\n';
    vertex<string, double> v = g[0];
    cout << v.val << '\n';
```

```

    if(g.count("Boston") == 0)
        cout << "Boston is not present" << '\n';
    v[1] = 100;
    for(auto it = g[0].begin(); it != g[0].end(); ++it){
        cout << g[0].val << "->" << g[it->first].val << ": " << it->second
<< '\n';
    }

    auto idx = g.get_index("Chicago");
    v[*idx] = 2.33;
    cout << g[0][1] << '\n';
    for(auto &p : g[0]){
        cout << g[0].val << "->" << g[p.first].val << ": " << p.second <<
'\n';
    }
    return 0;
}

```

Output

```

3
New York
Boston is not present
New York->Seattle: 100
100
New York->Chicago: 2.33
New York->Seattle: 100

```

Complexity

Constant time

end

```

typename unordered_map<uint32_t,Edge>::iterator end()

```

Returns iterator to end of edges originating from the vertex which it represents.

Parameters

None

Return Value

Iterator to end hashmap of index of target vertex and the edge weight.

Example

```

#include "graph_matrix.h"
#include <iostream>
#include <cassert>
using namespace graphmatrix;

int main(){
    graph<string, double> g;
    g.push_back("New York");
    g.push_back("Seattle");
    g.push_back("Chicago");

    // size of the graph
    cout << g.size() << '\n';
    vertex<string, double> v = g[0];
    cout << v.val << '\n';
    if(g.count("Boston") == 0)
        cout << "Boston is not present" << '\n';
    v[1] = 100;
    for(auto it = g[0].begin(); it != g[0].end(); ++it){
        cout << g[0].val << "->" << g[it->first].val << ": " << it->second
    << '\n';
    }

    auto idx = g.get_index("Chicago");
    v[*idx] = 2.33;
    cout << g[0][1] << '\n';
    for(auto &p : g[0]){
        cout << g[0].val << "->" << g[p.first].val << ": " << p.second <<
'\n';
    }
    return 0;
}

```

Output

```

3
New York
Boston is not present
New York->Seattle: 100
100
New York->Chicago: 2.33

```

New York->Seattle: 100

Complexity

Constant time

operator[]

Edge& operator[](uint32_t i)

Returns the edge from the current vertex to the given index. If an edge is not present then default constructor of “Edge” is invoked and the corresponding result’s reference is returned. Since it returns reference, it can be used to update the edge values as well.

Parameters

i	index to the vertex to which edge is being requested
---	------------------------------------------------------

Return Value

Reference to the value of the edge.

Exception Safety

If the index of vertex is out of range of the graph, then it throws vertex_out_of_range_error().

Example

```
#include "graph_matrix.h"
#include <iostream>
#include <cassert>
using namespace graphmatrix;

int main(){
    graph<string, double> g;
    g.push_back("New York");
    g.push_back("Seattle");
    g.push_back("Chicago");

    // size of the graph
    cout << g.size() << '\n';
    vertex<string, double> v = g[0];
    cout << v.val << '\n';
    if(g.count("Boston") == 0)
```

```

        cout << "Boston is not present" << '\n';
    v[1] = 100;
    for(auto it = g[0].begin(); it != g[0].end(); ++it){
        cout << g[0].val << "->" << g[it->first].val << ": " << it->second
    << '\n';
    }

    auto idx = g.get_index("Chicago");
    v[*idx] = 2.33;
    cout << g[0][1] << '\n';
    for(auto &p : g[0]){
        cout << g[0].val << "->" << g[p.first].val << ": " << p.second <<
'\n';
    }
    return 0;
}

```

Output

```

3
New York
Boston is not present
New York->Seattle: 100
100
New York->Chicago: 2.33
New York->Seattle: 100

```

Complexity

Constant amortized time, linear in size of graph in worst case.

operator =

```
void operator =(const T& newval)
```

Update the value of the current vertex

Parameters

newval	new value of the vertex to be updated
--------	---------------------------------------

Return Value

None

Exception Safety

If the vertex belongs to “graph” class, then it throws `duplicate_vertex_error()` if the value already exists for a different vertex.

Example

```
#include "graph_matrix.h"
#include <iostream>
#include <cassert>
using namespace graphmatrix;

int main(){
    graph<string, double> g;
    g.push_back("New York");
    g.push_back("Seattle");
    g.push_back("Chicago");

    // size of the graph
    cout << g.size() << '\n';
    vertex<string, double> v = g[0];
    cout << v.val << '\n';
    if(g.count("Boston") == 0)
        cout << "Boston is not present" << '\n';
    v[1] = 100;
    for(auto it = g[0].begin(); it != g[0].end(); ++it){
        cout << g[0].val << "->" << g[it->first].val << ": " << it->second
<< '\n';
    }

    auto idx = g.get_index("Chicago");
    v[*idx] = 2.33;
    cout << g[0][1] << '\n';
    for(auto &p : g[0]){
        cout << g[0].val << "->" << g[p.first].val << ": " << p.second <<
'\n';
    }
    return 0;
}
```

Output

```
3
New York
Boston is not present
New York->Seattle: 100
100
New York->Chicago: 2.33
New York->Seattle: 100
```

Complexity

Constant time for unordered_graph's vertex, $O(\log n)$ time for graph where vertex values support "==" and "<" operator. $O(n)$ for graph where vertex values only support "==" operator, where n is the number of vertices in the graph.

unordered_graph

```
template <class T, NumericType Edge> class unordered_graph
```

Overview

unordered_graph is a container that stores a set of vertices and edges between them. the types of vertices and edges are templated so any compatible type be used.

The definition of compatibility is the following:

- Edge type should be numeric, this means that it should support addition, multiplication, division, and subtraction. This is used by graph algorithm library to implement various algorithm on top of the representation.

Note: In contrast to graph class, this allows duplicate nodes and does not have lookup table.

Constructors

<i>default(1):</i>	unordered_graph()
<i>initializer_list(2):</i>	unordered_graph(const initializer_list &inp);
<i>copy(3):</i>	unordered_graph(const graph &g)
<i>move(4):</i>	unordered_graph(graph &&g)
<i>fill(5)</i>	unordered_graph(uint32_t n, const T& val = T())

(1) Empty graph constructor

- Constructs an empty graph with no vertex and no edges

(2) Initializer list constructor

- Constructs a graph with copy of elements in the initializer list in the same order while eliminating duplicates.

(3) Copy graph constructor

- Constructs a graph with copy of each vertex and edge in graph g

(4) Move graph constructor

- Constructor that acquires the vertices and edges of g. No elements are constructed, their ownership is directly transferred.

(5)

- Constructs a graph with vertices values specified in the initializer list in the given order.

Parameters

n	number of vertices in the graph
val	value of each of the n vertices in the graph
inp	list of vertex values
g	another graph object of same type

Example

```
#include "graph_matrix.h"
#include <iostream>
using namespace graphmatrix;

int main(){
    graph<int, int> empty_graph;
    graph<int, int> g2{1,2,3};
    auto g4 = g2;
    auto g5 = move(g1);
    std::cout << g1[5].val << '\n';
    return 0;
}
```

Output

12

Complexity

Constant for default(1) and move(4) constructors. $O(n)$ where n is the size of nodes for initializer list(2) constructor and fill (5) constructor. Linear in number of vertices and edge for copy(3) constructor.

Destructor

`~unordered_graph()`

This calls `allocator_traits::destroy` on each of the vertices and edges, and deallocates all the storage used by the graph

Complexity

Linear in number of vertices + number of edges in the graph.

push_back

```
uint32_t push_back(const T &val)
```

Pushes the new vertex to the end of the graph and returns the index of the new vertex inserted.

Parameters

val	value of the vertex to be inserted
-----	------------------------------------

Return Value

Index of the vertex that is newly inserted.

Example

```
#include "graph_matrix.h"
#include <iostream>
#include <cassert>
using namespace graphmatrix;

int main(){
    unordered_graph<string, double> g;
    g.push_back("New York");
    g.push_back("Seattle");
    g.push_back("Chicago");

    auto &nodes = g.nodes();
    for(auto& v : nodes) cout << v << ' ';
    cout << '\n';

    g.add_edge(0,1,10.4);
    cout << g[0][1] << '\n';
    g.erase_edge(0,1);
    cout << g[0][1] << '\n';

    return 0;
}
```


Output

```
New York Seattle Chicago
10.4
0
```

Complexity

$O(1)$ amortized time, worst case $O(n)$ where n is the number of vertices in the graph.

add_edge

```
void add_edge(uint32_t src, uint32_t target, const Edge &weight)
```

Adds a directed edge from vertex with index `src` to vertex with index `target` with edge weight as specified in the parameters

Parameters

src	index of source vertex
target	index of target index
weight	weight of the edge to be added/updated

Return Value

None

Exception Safety

If the index of vertex is out of range of the graph, then it throws `vertex_out_of_range_error()`.

Example

```
#include "graph_matrix.h"
#include <iostream>
#include <cassert>
using namespace graphmatrix;

int main(){
    unordered_graph<string, double> g;
    g.push_back("New York");
    g.push_back("Seattle");
    g.push_back("Chicago");
```

```

    auto &nodes = g.nodes();
    for(auto& v : nodes) cout << v << ' ';
    cout << '\n';

    g.add_edge(0,1,10.4);
    cout << g[0][1] << '\n';
    g.erase_edge(0,1);
    cout << g[0][1] << '\n';

    return 0;
}

```

Output

```

New York Seattle Chicago
10.4
0

```

Complexity

$O(1)$ amortized time complexity, $O(n^2)$ worst case time.

erase_edge

```
void erase_edge(uint32_t src, uint32_t target)
```

Removes the edge from vertex with index src to vertex with index target

Parameters

src	index of source vertex
target	index of target index

Return Value

None

Exception Safety

If the index of vertex is out of range of the graph, then it throws `vertex_out_of_range_error()`.

Example

```

#include "graph_matrix.h"
#include <iostream>
#include <cassert>
using namespace graphmatrix;

int main(){
    unordered_graph<string, double> g;
    g.push_back("New York");
    g.push_back("Seattle");
    g.push_back("Chicago");

    auto &nodes = g.nodes();
    for(auto& v : nodes) cout << v << ' ';
    cout << '\n';

    g.add_edge(0,1,10.4);
    cout << g[0][1] << '\n';
    g.erase_edge(0,1);
    cout << g[0][1] << '\n';

    return 0;
}

```

Output

```

New York Seattle Chicago
10.4
0

```

Complexity

$O(1)$ amortized time complexity, $O(n^2)$ worst case time.

add_undirected_edge

```
void add_undirected_edge(uint32_t src, uint32_t target, const Edge &weight)
```

Adds a undirected edge between vertex with index src and vertex with index target with edge weight as specified in the parameters

Parameters

src	index of source vertex
target	index of target index
weight	weight of the edge to be added/updated

Return Value

None

Exception Safety

If the index of vertex is out of range of the graph, then it throws `vertex_out_of_range_error()`.

Example

```
#include "graph_matrix.h"
#include <iostream>
#include <cassert>
using namespace graphmatrix;

int main(){
    unordered_graph<string, double> g;
    g.push_back("New York");
    g.push_back("Seattle");
    g.push_back("Chicago");

    auto &nodes = g.nodes();
    for(auto& v : nodes) cout << v << ' ';
    cout << '\n';

    g.add_undirected_edge(0,1,10.4);
    cout << g[0][1] << '\n';
    g.erase_undirected_edge(1,0);
    cout << g[0][1] << '\n';

    return 0;
}
```

Output

```
New York Seattle Chicago
10.4
0
```

Complexity

$O(1)$ amortized time complexity, $O(n^2)$ worst case time.

erase_undirected_edge

```
void erase_undirected_edge(uint32_t src, uint32_t target)
```

Removes the edge between vertex with index src and vertex with index target

Parameters

src	index of source vertex
target	index of target index

Return Value

None

Exception Safety

If the index of vertex is out of range of the graph, then it throws vertex_out_of_range_error().

Example

```
#include "graph_matrix.h"
#include <iostream>
#include <cassert>
using namespace graphmatrix;

int main(){
    unordered_graph<string, double> g;
    g.push_back("New York");
    g.push_back("Seattle");
    g.push_back("Chicago");

    auto &nodes = g.nodes();
    for(auto& v : nodes) cout << v << ' ';
    cout << '\n';

    g.add_undirected_edge(0,1,10.4);
    cout << g[0][1] << '\n';
    g.erase_undirected_edge(1,0);
```

```

    cout << g[0][1] << '\n';

    return 0;
}

```

Output

```

New York Seattle Chicago
10.4
0

```

Complexity

$O(1)$ amortized time complexity, $O(n^2)$ worst case time.

nodes

```
const vector<T>& nodes() const
```

Gets reference to the vector containing values of the nodes. The reference is constant and hence it cannot be used to update the vertex values.

Parameters

None

Return Value

Reference to vector containing values of vertices of the graph

Example

```

#include "graph_matrix.h"
#include <iostream>
#include <cassert>
using namespace graphmatrix;

int main(){
    unordered_graph<string, double> g;
    g.push_back("New York");
    g.push_back("Seattle");
    g.push_back("Chicago");

    auto &nodes = g.nodes();
    for(auto& v : nodes) cout << v << ' ';
    cout << '\n';
}

```

```
    return 0;
}
```

Output

```
New York Seattle Chicago
```

Complexity

Constant time.

size

```
inline size_t size()
```

```
const inline size_t size() const
```

Gets the size of the graph which is equal to number of vertices in the graph.

Parameters

None

Return Value

Size of the graph

Example

```
#include "graph_matrix.h"
#include <iostream>
#include <cassert>
using namespace graphmatrix;

int main(){
    unordered_graph<string, double> g;
    g.push_back("New York");
    g.push_back("Seattle");
    g.push_back("Chicago");

    cout << g.size() << '\n';

    g[1][0] = 12.33;
    g[1][2] = 54.22;
    vertex<string, double> v = g[1];
    for(auto &p : v){
```

```

        cout << g[1].val << "->" << g[p.first].val << ": " << p.second <<
        '\n';
    }

    return 0;
}

```

Output

```

3
Seattle->Chicago: 54.22
Seattle->New York: 12.33

```

Complexity

Constant time.

operator[]

```
vertex<T, Edge> operator[](uint32_t i)
```

Gets the vertex object that contains information about the vertex like its value and edges originating from it. It can also be used to update the edge value using the subscript operator overloaded in the vertex class.

Parameters

i	index of the vertex to get information about
---	----------------------------------------------

Return Value

an object of class vertex<T, Edge>

Exception Safety

If the index of vertex is out of range of the graph, then it throws vertex_out_of_range_error().

Example

```

#include "graph_matrix.h"
#include <iostream>
#include <cassert>
using namespace graphmatrix;

int main(){

```



```

unordered_graph<string, double> g;
g.push_back("New York");
g.push_back("Seattle");
g.push_back("Chicago");

cout << g.size() << '\n';

g[1][0] = 12.33;
g[1][2] = 54.22;
vertex<string, double> v = g[1];
for(auto &p : v){
    cout << g[1].val << "->" << g[p.first].val << ": " << p.second <<
'\n';
}

return 0;
}

```

Output

```

3
Seattle->Chicago: 54.22
Seattle->New York: 12.33

```

Complexity

Constant time
