# Documentation of Graph Explorer

## 

## 

## 

## 

## Table of Contents

[**Documentation of Graph Explorer**](#_wckybv9jws79) **0**

[Table of Contents](#_q8i1h1w2b4q6) 1

[graph](#_fmfm9ideisqt) 6

[Overview](#_kqx8hjkr858l) 6

[Constructors](#_mha065dn121k) 6

[Parameters](#_o2nlus0fy34) 7

[Example](#_k8dgg1nzdt4l) 7

[Output](#_h5j661sxxw3y) 7

[Complexity](#_5bv7zquz6lpt) 7

[Destructor](#_6ajvymhe0mhg) 7

[Complexity](#_rz7dir3ijuo5) 8

[push\_back](#_krtx3uur4h4d) 8

[Parameters](#_vbimx2u7od7d) 8

[Return Value](#_wqmhod951ryk) 8

[Example](#_ffgchhu6bzq) 8

[Output](#_1wsbag1ilirh) 9

[Complexity](#_iuvumnxr4jno) 9

[add\_edge](#_qog8umhczths) 9

[Parameters](#_cs4aq46nz7ad) 9

[Return Value](#_ypt6taxys74c) 10

[None](#_gy33swl3s5r) 10

[Exception Safety](#_u52s59yardz7) 10

[Example](#_ylz2vaqg52mk) 10

[Output](#_t0wdsxwivdmn) 10

[Complexity](#_6baez8eefwwc) 11

[erase\_edge](#_j6a85o49rtr1) 11

[Parameters](#_kzax85gbctqm) 11

[Return Value](#_9xutrkdw3t90) 11

[None](#_a8rcz8xy2h2y) 11

[Exception Safety](#_8lpmkrt3c7g2) 11

[Example](#_qbv5z87rcam) 11

[Output](#_3z8cvb64mctu) 11

[Complexity](#_o5a3b51q3qay) 12

[add\_undirected\_edge](#_4u5eeed5sj4q) 12

[Parameters](#_ci6tuvk9082y) 12

[Return Value](#_hubrhisyhagd) 13

[Exception Safety](#_e21x57ac5uap) 13

[Example](#_boj58j5pndv7) 13

[Output](#_kyx6qwreewmu) 13

[Complexity](#_vpjp89ld37zf) 14

[erase\_undirected\_edge](#_ov7y6u3coit3) 14

[Parameters](#_5oyhnrf5mi9p) 14

[Return Value](#_jkr80bg8uh2h) 14

[Exception Safety](#_99ci06lnd3ca) 14

[Example](#_wnh5e6ek3o5v) 14

[Output](#_fs671qyqz1lx) 14

[Complexity](#_ouojzt9vxzva) 15

[nodes](#_e1r5sgplyj85) 15

[Parameters](#_iph38bh75dlj) 15

[Return Value](#_ckgc55b7pl7r) 15

[Example](#_1acd51t8djyq) 15

[Output](#_u9bk4qz1r2n0) 16

[Complexity](#_35z0eiah7kln) 16

[size](#_sjrzx71hwh6) 16

[Parameters](#_3vyxgqhy1c8d) 16

[Return Value](#_78vh0qicqrjc) 16

[Example](#_esw7kf9nz962) 16

[Output](#_hv70vl28f5nq) 16

[Complexity](#_cz9a5hn053na) 17

[operator[]](#_sgovwggj7rx8) 17

[Parameters](#_gh8t4k9gbkn7) 18

[Return Value](#_wnj6cnlj33lk) 18

[Exception Safety](#_ewce0nbmflrv) 18

[Example](#_x6p5y6651d3s) 18

[Output](#_pf8mkaffs0co) 18

[Complexity](#_hwp36j2ch6ld) 19

[count](#_e9ossohl0r10) 19

[Parameters](#_3icxinh30e1h) 19

[Return Value](#_n43isxx75a95) 19

[Example](#_1k4kkp410ki7) 19

[Output](#_f1fq9s9558a9) 19

[Complexity](#_3lisgu81rdl) 20

[get\_index](#_m501mtr6hdeb) 20

[Parameters](#_czrl8lvd2zgi) 20

[Return Value](#_e1t0askwib08) 20

[Example](#_jej15ngek7d9) 20

[Output](#_s3n7n7g9r2kw) 21

[Complexity](#_bg67sf18zsmd) 21

[vertex](#_wv83f5vh9wjj) 21

[Methods](#_u13w68b51541) 22

[begin](#_lvghl769b7um) 22

[Parameters](#_xnxk9tklym32) 22

[Return Value](#_zi8x9yjqvxte) 22

[Example](#_qrl83wrjoq4) 22

[Output](#_tj2aahv8vsim) 22

[Complexity](#_m7gos3c1z3yr) 23

[end](#_a9krxw1d7oig) 23

[Parameters](#_mkz8mry76jku) 23

[Return Value](#_wmpjy6k8czq9) 23

[Example](#_uosaxllqv9wr) 23

[Output](#_e5hmlvebh44t) 24

[Complexity](#_muztq6qym4ur) 25

[operator[]](#_qywv24bjd007) 25

[Parameters](#_hysjtiyem0kf) 25

[Return Value](#_oa8f9vineha6) 25

[Exception Safety](#_onynuiyok55c) 25

[Example](#_5b64tnpvj8f9) 25

[Output](#_ln24sxh6qv2q) 25

[Complexity](#_g3tpj4w81jh3) 26

[operator =](#_n1a2bcmcbbwc) 26

[Parameters](#_royastnibqo9) 26

[Return Value](#_tqoxns7uu20l) 26

[None](#_bb36ypg09vmz) 27

[Exception Safety](#_qhveq3ek1eni) 27

[Example](#_1ut87tj5ulyx) 27

[Output](#_55y92admhac7) 27

[Complexity](#_2zfs7d25q9y7) 28

[unordered\_graph](#_d13tgb92d015) 29

[Overview](#_w03n22fqic5q) 29

[Constructors](#_4z8r0cor4774) 29

[Parameters](#_dxzqshfdvtgs) 29

[Example](#_sun1u6fyhpmw) 30

[Output](#_wged8hj1ocql) 30

[Complexity](#_febl4p2o2saw) 30

[Destructor](#_ihdw9bf9471l) 30

[Complexity](#_hh7mmded30my) 30

[push\_back](#_qtozsh1owto) 31

[Parameters](#_29s3t7m2soxl) 31

[Return Value](#_qa1z3mh1f0bc) 31

[Example](#_q1u7auifiyv1) 31

[Output](#_wursfd3y6vu) 31

[Complexity](#_sgeba0ny9996) 32

[add\_edge](#_1t64tr4gac3j) 32

[Parameters](#_9ycdzlhpicsp) 32

[Return Value](#_k2kjlt7tibh3) 32

[None](#_f3m8g7d5mgw) 32

[Exception Safety](#_a772ahow6t2a) 32

[Example](#_7erq8rbh2sht) 32

[Output](#_5x7uj2g8l41h) 32

[Complexity](#_2bpcdqbd3acj) 33

[erase\_edge](#_17s89al24kl6) 33

[Parameters](#_j6m1419u2iw5) 33

[Return Value](#_23i3d963v3gj) 33

[None](#_18met5j61u5b) 33

[Exception Safety](#_6esrdsnawtr4) 33

[Example](#_hcen7jl8xut4) 33

[Output](#_1j1dffwpp3kd) 34

[Complexity](#_tqze9j8ji627) 34

[add\_undirected\_edge](#_soqazfxcniua) 34

[Parameters](#_ejdtdmxdajzp) 35

[Return Value](#_grn66x1q9ylx) 35

[Exception Safety](#_dsy4o74dqpoq) 35

[Example](#_ag8lj8ca9vgz) 35

[Output](#_xwjde9d06vf0) 35

[Complexity](#_nnetq9p5toa) 36

[erase\_undirected\_edge](#_q4idgtheksho) 36

[Parameters](#_udzeos7qozid) 36

[Return Value](#_ud4pnswo6u81) 36

[Exception Safety](#_j55q3jhcj7y2) 36

[Example](#_r0jef6w2yu63) 36

[Output](#_l9b614fb2ybw) 36

[Complexity](#_lcw3qdtf55fp) 37

[nodes](#_acu41giyvqdr) 37

[Parameters](#_o7ymtpaetkg0) 37

[Return Value](#_2t50zkwxs9eu) 37

[Example](#_p1e188iezcmv) 37

[Output](#_3kf746znx1g) 38

[Complexity](#_uqc450xv9b8i) 38

[size](#_6kn20l4ylydy) 38

[Parameters](#_ey1609u9rxc0) 38

[Return Value](#_c9vghzuwtz8z) 38

[Example](#_ngg0a9bw05p9) 38

[Output](#_1n4vwbi6aezd) 38

[Complexity](#_ialaocy0xclo) 39

[operator[]](#_sdpj8fxmdky7) 39

[Parameters](#_uvrcs34mve70) 39

[Return Value](#_jhsmrgsixp5x) 39

[Exception Safety](#_wgzn52o61a7q) 39

[Example](#_bwzojnmxgeyg) 39

[Output](#_jrtgimu4wg3i) 39

[Complexity](#_qnqmzhf1r3ji) 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 templatized 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

|  |  |
| --- | --- |
| *default(1):* | graph() |
| *initializer\_list(2):* | graph(const initializer\_list &inp); |
| *copy(3):* | graph(const graph &g) |
| *move(4):* | 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 logn) 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(logn) amortized time and O(n logn) 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(logn) 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 templatized 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

|  |  |
| --- | --- |
| *default(1):* | unordered\_graph() |
| *initializer\_list(2):* | unordered\_graph(const initializer\_list &inp); |
| *copy(3):* | unordered\_graph(const graph &g) |
| *move(4):* | unordered\_graph(graph &&g) |
| *fill(5)* | 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