Graph

Generated by Doxygen 1.8.11

ii CONTENTS

Contents

1	Todo	o List	2
2	Bug	List	2
3	Data	a Structure Index	2
	3.1	Data Structures	2
4	File	Index	2
	4.1	File List	2
5	Data	a Structure Documentation	3
	5.1	deg_t Struct Reference	3
		5.1.1 Detailed Description	3
		5.1.2 Field Documentation	3
	5.2	graph_t Struct Reference	4
		5.2.1 Detailed Description	4
		5.2.2 Field Documentation	4
	5.3	gvertex_t Struct Reference	5
		5.3.1 Detailed Description	5
		5.3.2 Field Documentation	6
	5.4	info_t Struct Reference	6
		5.4.1 Detailed Description	6
		5.4.2 Field Documentation	6
	5.5	queue_t Struct Reference	7
		5.5.1 Detailed Description	7
		5.5.2 Field Documentation	7
	5.6	stack_t Struct Reference	8
		5.6.1 Detailed Description	8
		5.6.2 Field Documentation	8

6	File	Documentation		
	6.1	graphs	.c File Reference	9
		6.1.1	Detailed Description	10
		6.1.2	Function Documentation	10
	6.2	graphs	.c	15
	6.3	graphs	h File Reference	18
		6.3.1	Detailed Description	19
		6.3.2	Enumeration Type Documentation	20
		6.3.3	Function Documentation	20
	6.4	graphs	.h	25
	6.5	queue.	c File Reference	26
		6.5.1	Detailed Description	27
		6.5.2	Function Documentation	27
	6.6	queue.	c	29
	6.7	queue.	h File Reference	30
		6.7.1	Detailed Description	31
		6.7.2	Function Documentation	31
	6.8	queue.	h	33
	6.9	stack.c	File Reference	34
		6.9.1	Detailed Description	34
		6.9.2	Function Documentation	34
	6.10	stack.c		37
	6.11	stack.h	File Reference	37
		6.11.1	Detailed Description	38
		6.11.2	Function Documentation	38
	6.12	stack.h		41
Inc	dex			43
				-

1 Todo List

Global graph_dot_output (graph_t *g, char *filename)

system call?

Class queue_t

base should'nt be accessible, see https://stackoverflow.com/questions/5368028/how-to-make-struct-

2 Bug List

Global graph_clone (graph_t *g)

sizeof(pt) is wrong!!

3 Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

deg_t	3
graph_t	4
gvertex_t	5
info_t	6
queue_t	7
stack_t	8

4 File Index

4.1 File List

Here is a list of all files with brief descriptions:

graphs.c Graph's basic operations implementation	9
graphs.h Graph definition and basic operations	18
queue.c Queue's basic operations implementation (using dynamic array)	26
queue.h Queue (using array) definition and basic operations	30

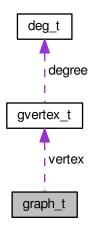
```
stack.c
       Stack's basic operations implementation (using dynamic array)
                                                                                                            34
       Stack definition and basic operations
                                                                                                            37
5 Data Structure Documentation
5.1 deg_t Struct Reference
#include <graphs.h>
Data Fields
    • int in
    · int out
5.1.1 Detailed Description
Definition at line 23 of file graphs.h.
5.1.2 Field Documentation
5.1.2.1 int in
indegree
Definition at line 24 of file graphs.h.
5.1.2.2 int out
outdegree
Definition at line 25 of file graphs.h.
The documentation for this struct was generated from the following file:
```

• graphs.h

5.2 graph_t Struct Reference

#include < graphs.h>

Collaboration diagram for graph_t:



Data Fields

- gtype_t type
- int n
- int m
- gvertex_t ** vertex

5.2.1 Detailed Description

Definition at line 34 of file graphs.h.

5.2.2 Field Documentation

5.2.2.1 int m

number of edges

Definition at line 37 of file graphs.h.

5.2.2.2 int n

number of vertices

Definition at line 36 of file graphs.h.

5.2.2.3 gtype_t type

graph type: undirected or directed

Definition at line 35 of file graphs.h.

5.2.2.4 gvertex_t** vertex

list of vertices

Definition at line 38 of file graphs.h.

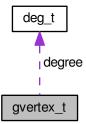
The documentation for this struct was generated from the following file:

• graphs.h

5.3 gvertex_t Struct Reference

```
#include <graphs.h>
```

Collaboration diagram for gvertex_t:



Data Fields

- int * adj_list
- int adj_list_len
- deg_t degree

5.3.1 Detailed Description

Definition at line 28 of file graphs.h.

```
5.3.2 Field Documentation
5.3.2.1 int* adj_list
adjacency list (list of vertices index)
Definition at line 29 of file graphs.h.
5.3.2.2 int adj_list_len
sizeof(int) * adj_list_len bytes is reserved for adj_list
Definition at line 30 of file graphs.h.
5.3.2.3 deg_t degree
indegree and outdegree
Definition at line 31 of file graphs.h.
The documentation for this struct was generated from the following file:
    • graphs.h
5.4 info_t Struct Reference
#include <graphs.h>
Data Fields
    int * pred
    int * dist
5.4.1 Detailed Description
Definition at line 41 of file graphs.h.
5.4.2 Field Documentation
5.4.2.1 int* dist
Definition at line 43 of file graphs.h.
5.4.2.2 int* pred
Definition at line 42 of file graphs.h.
The documentation for this struct was generated from the following file:
```

• graphs.h

5.5 queue_t Struct Reference

```
#include <queue.h>
```

Data Fields

- size_t width
- int front
- · int count
- void ** base
- int max_size

5.5.1 Detailed Description

Abstract queue using array.

Todo base should'nt be accessible, see https://stackoverflow.com/questions/5368028/how-to-make-struc

Definition at line 21 of file queue.h.

5.5.2 Field Documentation

5.5.2.1 void** base

pointer to the array

Definition at line 25 of file queue.h.

5.5.2.2 int count

count element amount

Definition at line 24 of file queue.h.

5.5.2.3 int front

front element index

Definition at line 23 of file queue.h.

5.5.2.4 int max_size

width * max_size bytes is reserved for the queue

Definition at line 26 of file queue.h.

```
5.5.2.5 size_t width
element size (in bytes)
Definition at line 22 of file queue.h.
The documentation for this struct was generated from the following file:
    • queue.h
5.6 stack_t Struct Reference
#include <stack.h>
Data Fields

    size_t width

    int top

    void ** base
    • int mem_size
5.6.1 Detailed Description
Abstract stack using dynamic array.
Definition at line 20 of file stack.h.
5.6.2 Field Documentation
5.6.2.1 void** base
pointer to the dynamic array
Definition at line 23 of file stack.h.
5.6.2.2 int mem_size
width * mem_size bytes is reserved for the dynamic array
Definition at line 24 of file stack.h.
5.6.2.3 int top
top element index
```

Definition at line 22 of file stack.h.

6 File Documentation 9

```
5.6.2.4 size_t width
```

element size (in bytes)

Definition at line 21 of file stack.h.

The documentation for this struct was generated from the following file:

· stack.h

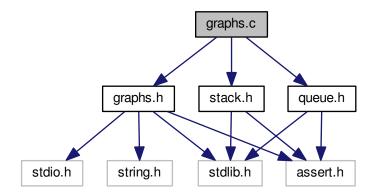
6 File Documentation

6.1 graphs.c File Reference

graph's basic operations implementation

```
#include "graphs.h"
#include "queue.h"
#include "stack.h"
```

Include dependency graph for graphs.c:



Functions

- graph_t * graph_create (int n, gtype_t type)
- void graph_destruct (graph_t *g)
- int is_adj (graph_t *g, int u, int v)
- void graph_add_edge (graph_t *g, int u, int v)
- void graph_adj_list_print (graph_t *g)
- graph_t * graph_clone (graph_t *g)
- void graph_delete_adj_ele (graph_t *g, int u, int v)
- void graph_dot_output (graph_t *g, char *filename)
- info_t * bfs (graph_t *g, int src)

6.1.1 Detailed Description

graph's basic operations implementation

Author

Firmin MARTIN

Version

0.1

Date

28/12/2017

Definition in file graphs.c.

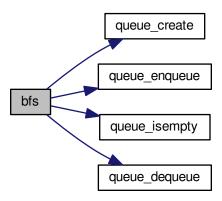
6.1.2 Function Documentation

6.1.2.1 info_t* bfs (graph_t * g, int src)

Definition at line 220 of file graphs.c.

```
00220
00221
              int* color = malloc(g->n * sizeof(int));
             int* pred = malloc(g > n * sizeof(int));
int* dist = malloc(g - > n * sizeof(int));
00222
00223
00224
              assert(color && pred && dist);
             for(int i = 0; i < g->n; i++) {
  color[i] = 0;
  pred[i] = -1;
  dist[i] = g->n + 1;
00225
00226
00227
00228
00229
00230
             queue_t* q = queue_create(sizeof(int), g->n);
00231
              assert(q);
             int i = src;
for (int j = 0; j < g->n; j++) {
    if (color[i] == 0) {
        color[i] = 1;
}
00232
00233
00234
00235
00236
                         dist[0];
                        pred[i] = -1;
00237
00238
                         queue_enqueue(q, &i);
                         while(!queue_isempty(q)) {
  int u = q>front;
  for(int k = 0; k < g->vertex[u]->degree.out; k++) {
00239
00240
00241
                                    if (color[k] == 0) {
    color[k] = 1;
00242
00243
                                          dist[k] = dist[u] + 1;
pred[k] = u;
00244
00245
00246
                                          queue_enqueue(q, &k);
00247
00248
                               queue_dequeue(q);
color[u] = 2;
00249
00250
00251
                         }
00252
00253
                   i = j;
00254
00255
              free(color);
00256
              info_t* info;
             info->dist = dist;
info->pred = pred;
return info;
00257
00258
00259
00260 }
```

Here is the call graph for this function:



6.1.2.2 void graph_add_edge (graph_t * g, int u, int v)

Add the edge (u, v) in the graph g. If g is an undigraph, (v, u) is also added.

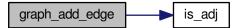
Parameters

g	graph
и	vertex index
V	vertex index

Definition at line 78 of file graphs.c.

```
00078
         gvertex_t *v_u = g->vertex[u], *v_v = g->vertex[v];
assert(u >= 0 && v >= 0 && u < g->n && v < g->n);
00079
08000
         assert(v_u->degree.out <= v_u->adj_list_len);
00082
          if (is_adj(g, u, v)) return;
00083
         if (v_u->degree.out == v_u->adj_list_len) {
             int* newlist = realloc(v_u->adj_list, sizeof(int) * (v_u->
00084
     adj_list_len + 10));
00085
            assert(newlist);
             v_u->adj_list = newlist;
v_u->adj_list_len += 10;
00086
00087
00088
00089
         v_u->adj_list[v_u->degree.out] = v;
00090
         v_u->degree.out++;
         v_v->degree.in++;
00091
         if (g->type == UNDIGRAPH && u != v) {
00092
00093
              assert(v_v->degree.out <= v_v->adj_list_len);
00094
              if (v_v->degree.out == v_v->adj_list_len) {
                 00095
00096
                 assert(newlist);
00097
                 v_v->adj_list = newlist;
00098
                 v_v->adj_list_len += 10;
00099
00100
              v_v->adj_list[v_v->degree.out] = u;
00101
             v_v->degree.out++;
00102
             v_u->degree.in++;
00103
00104
         g->m++;
00105 }
```

Here is the call graph for this function:



```
6.1.2.3 void graph_adj_list_print ( graph_t * g )
```

Print graph's adjacency list representation

Parameters

```
g graph
```

Definition at line 111 of file graphs.c.

6.1.2.4 graph_t* graph_clone (graph_t * g)

Clone a graph

Parameters



Returns

graph cloned

Bug sizeof(pt) is wrong!!

Definition at line 131 of file graphs.c.

```
00131
00132
          graph_t* new_g;
00133
          new_g = (graph_t*) malloc(sizeof(graph_t));
00134
          if (new_g != NULL) {
00135
              new_g->type = g->type;
              new_g->n=g->n;
00136
              new_g->m = g->m;
00137
00138
              new_g->vertex = (gvertex_t**) malloc(sizeof(g->vertex));
00139
              assert(new_g->vertex);
              for(int i = 0; i < g->n; i++) {
   new_g->vertex[i] = (gvertex_t*) malloc(sizeof()
00140
00141
      gvertex_t));
00142
                  assert(q->vertex[i]);
                  new_g->vertex[i]->adj_list = (int*) malloc(sizeof(g->
      vertex[i]->adj_list));
00144
                  assert(new_g->vertex[i]->adj_list);
00145
                  new_g->vertex[i]->adj_list_len = g->vertex[i]->
      adj_list_len;
00146
                  new_q->vertex[i]->degree.in = q->vertex[i]->
      degree.in;
00147
                  new_g->vertex[i]->degree.out = g->vertex[i]->
      degree.out;
00148
00149
00150
          return new_q;
00151 }
```

6.1.2.5 graph t* graph_create (int n, gtype t type)

Create a graph initialized as a forest with n vertices

Parameters

n	number of vertices
type	type of graph (digraph, undigraph)

Returns

return a graph initialized as a forest

Definition at line 19 of file graphs.c.

```
00019
00020
            graph_t* g;
00021
            g = (graph_t*) malloc(sizeof(graph_t));
00022
            assert(g);
00023
            if (g != NULL) {
                 g->type = type; /* initialize type of graph */
g->n = n; /* initialize number of vertices */
g->m = 0; /* g is a forest => 0 edge */
00025
00026
00027
                 g->vertex = (gvertex_t**) malloc(sizeof(gvertex_t*) * n);
00028
                 assert(g->vertex);
                 for(int i = 0; i < g->n; i++) {
    g->vertex[i] = (gvertex_t*) malloc(sizeof(gvertex_t));
00029
00030
00031
                     assert(g->vertex[i]);
00032
                     /* initialize an array with size 5 by default */
00033
                     g->vertex[i]->adj_list = (int*) malloc(sizeof(int) * 5);
00034
                     assert(g->vertex[i]->adj_list);
                     g->vertex(i)->adj_list_len = 5;
/* g is a forest => deg_in(i) = deg_out(i) = 0 */
00035
00036
00037
                     g->vertex[i]->degree.in = 0;
00038
                     g->vertex[i]->degree.out = 0;
00039
                }
00040
            return q;
00041
00042 }
```

6.1.2.6 void graph_delete_adj_ele (graph_t * g, int u, int v)

Delete the edge (u,v) in graph g. If g is an undigraph, (v,u) is also removed.

Parameters

g	graph
и	vertex index
V	vertex index

Definition at line 159 of file graphs.c.

```
00159
              gvertex_t *v_u = g->vertex[u], *v_v = g->vertex[v];
assert(v_u && v_v);
00160
00161
00162
               int flag = 0;
00163
               for (int i = 0; i < v_u->degree.out; i++) {
00164
                      if(v_u->adj_list[i] == v) {
00165
                           flag = 1;
                           for (int j = i; j < v_u->degree.out - 1; j++) {
    v_u->adj_list[j] = v_u->adj_list[j + 1];
00166
00167
00168
00169
                           v_u->degree.out--;
00170
00171
              if (g->type == UNDIGRAPH && flag) {
   for (int i = 0; i < v_v->degree.out; i++) {
      if(v_v->adj_list[i] == u) {
        for (int j = i; j < v_v->degree.out - 1; j++) {
            v_v->adj_list[j] = v_v->adj_list[j + 1];
      }
}
00172
00173
00174
00175
00176
00177
00178
                                  v_v->degree.out--;
00179
                           }
00180
                     }
00181
00182
               if(flag) g->m--;
00183 }
```

6.1.2.7 void graph_destruct (graph_t * g)

Free a graph

Parameters

```
g a graph
```

Definition at line 48 of file graphs.c.

6.1.2.8 void graph_dot_output (graph_t * g, char * filename)

Output the graph g in dot format (filename.dot) and use dot compile it to filename.ps .

Parameters

g	graph
filename	filename without any extension

6.2 graphs.c 15

Todo system call?

Definition at line 192 of file graphs.c.

```
00192
              FILE* pfile;
00193
00194
              pfile = fopen(filename, "w");
00195
               if (pfile == NULL) perror ("Error opening file");
00196
               if (g->type == DIGRAPH) {
                    fprintf(pfile, "digraph g {\nnode [shape=\"circle\"];\n");
for(int i = 0; i < g->n; i++) {
00197
00198
                          fif (g>vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
for(int j = 0; j < g->vertex[i]->degree.out; j++) {
    fprintf(pfile, "%d -> %d;\n", i, g->vertex[i]->adj_list[j]);
00199
00200
00201
00202
00203
                    }
              } else if (g->type == UNDIGRAPH) {
  fprintf(pfile, "graph g {\nnode [shape=\"circle\"];\n");
  for(int i = 0; i < g->n; i++) {
00204
00205
00206
                          if(g->vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
                          for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
    if(i <= g->vertex[i]->adj_list[j]) {
        fprintf(pfile, "%d -- %d;\n", i, g->vertex[i]->
00208
00209
00210
adj_list[j]);
00212
00213
                    }
00214
00215
              fprintf(pfile, "\n");
00216
              fclose(pfile);
00217
              //system(strcat(strcat(strcat(strcat("dot ", filename),"-Tps -o "), filename),".ps"));
00218 }
```

6.1.2.9 int is_adj (graph_t * g, int u, int v)

Determinate if v is in the adjacency list of u in graph g

Parameters

g	graph
и	vertex index
V	vertex index

Definition at line 63 of file graphs.c.

6.2 graphs.c

```
g->type = type; /* initialize type of graph */
                                    /* initialize number of vertices */
/* g is a forest => 0 edge */
00025
                 g->m = 0;
00026
                 g->vertex = (gvertex_t**) malloc(sizeof(gvertex_t*) * n);
00027
                 assert(g->vertex);
for(int i = 0; i < g->n; i++) {
   g->vertex[i] = (gvertex_t*) malloc(sizeof(gvertex_t));
00028
00029
00030
00031
                      assert(g->vertex[i]);
00032
                       /\star initialize an array with size 5 by default \star/
                      g->vertex[i]->adj_list = (int*) malloc(sizeof(int) * 5);
assert(g->vertex[i]->adj_list);
00033
00034
                      g->vertex(i]->adj_list_len = 5;
/* g is a forest => deg_in(i) = deg_out(i) = 0 */
00035
00036
00037
                      g->vertex[i]->degree.in = 0;
00038
                      g->vertex[i]->degree.out = 0;
00039
                 }
00040
00041
            return q;
00042 }
00043
00048 void graph_destruct(graph_t* g) {
           for (int i = 0; i < g->n; i++) {
    free(g->vertex[i]->adj_list);
00049
00050
00051
                 free(g->vertex[i]);
00052
00053
            free (g->vertex);
00054
            free(g);
00055 }
00056
00063 int is_adj(graph_t* g, int u, int v) {
00064
            gvertex_t* v_u = g->vertex[u];
00065
            assert (v_u);
00066
            for(int i = 0; i < v_u->degree.out; i++) {
00067
                 if(v_u->adj_list[i] == v) return 1;
00068
00069
            return 0:
00070 }
00071
00078 void graph_add_edge(graph_t* g, int u, int v) {
            gvertex_t *v_u = g->vertex[u], *v_v = g->vertex[v];
assert(u >= 0 && v >= 0 && u < g->n && v < g->n);
00079
00080
00081
            assert(v_u->degree.out <= v_u->adj_list_len);
            if (is_adj(g, u, v)) return;
if (v_u->degree.out == v_u->adj_list_len) {
00082
00083
                 int* newlist = realloc(v_u->adj_list, sizeof(int) * (v_u->
00084
       adj_list_len + 10));
00085
                assert(newlist);
00086
                 v_u->adj_list = newlist;
v_u->adj_list_len += 10;
00087
00088
00089
            v_u->adj_list[v_u->degree.out] = v;
00090
            v_u->degree.out++;
00091
            v_v->degree.in++;
            if (g->type == UNDIGRAPH && u != v) {
  assert(v_v->degree.out <= v_v->adj_list_len);
  if (v_v->degree.out == v_v->adj_list_len) {
00092
00093
00094
                      int* newlist = realloc(v_v->adj_list, sizeof(int) * (v_v->adj_list_len + 10));
00095
00096
                      assert(newlist);
                      v_v->adj_list = newlist;
v_v->adj_list_len += 10;
00097
00098
00099
                 v_v->adj_list[v_v->degree.out] = u;
00100
00101
                 v_v->degree.out++;
00102
                 v_u->degree.in++;
00103
            g->m++;
00104
00105 }
00106
00111 void graph_adj_list_print(graph_t* g) {
           if (g->type == DIGRAPH) printf("type : digraph\n");
else if (g->type == UNDIGRAPH) printf("type : undigraph\n");
00113
00114
            printf("n=%d, m=%d\n", g->n, g->m);
            for(int i = 0; i < g->n; i++) {
   printf("[%d] ", i);
   if (g->vertex[i]->degree.out > 0) printf("-> ");
00115
00116
00117
                 for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
    printf("[%d]", g->vertex[i]->adj_list[j]);
00118
00119
00120
                 printf("\n");
00121
00122
            }
00123 }
00124
00131 graph_t* graph_clone(graph_t* g) {
            graph_t* new_g;
new_g = (graph_t*) malloc(sizeof(graph_t));
00132
00133
            if (new_g != NULL) {
00134
00135
                 new_g->type = g->type;
```

6.2 graphs.c 17

```
00136
                 new_q->n = q->n;
                 new_g->m = g->m;
00137
00138
                 new_g->vertex = (gvertex_t**) malloc(sizeof(g->vertex));
00139
                 assert(new_g->vertex);
                for(int i = 0; i < g->n; i++) {
    new_g->vertex[i] = (gvertex_t*) malloc(sizeof(
00140
00141
00142
                     assert(q->vertex[i]);
00143
                      new_g->vertex[i]->adj_list = (int*) malloc(sizeof(g->
       vertex[i]->adj_list));
00144
                     assert(new_g->vertex[i]->adj_list);
                      new_g->vertex[i]->adj_list_len = g->vertex[i]->
00145
       adi list len;
00146
                     new_g->vertex[i]->degree.in = g->vertex[i]->
       degree.in;
00147
                     new_g->vertex[i]->degree.out = g->vertex[i]->
       degree.out;
           }
00148
00149
00150
            return new_g;
00151 }
00152
00159 void graph_delete_adj_ele(graph_t* g, int u, int v) {
00160
           gvertex_t *v_u = g->vertex[u], *v_v = g->vertex[v];
assert(v_u && v_v);
00161
00162
            int flag = 0;
00163
            for (int i = 0; i < v_u->degree.out; i++) {
00164
                 if(v_u->adj_list[i] == v) {
00165
                      flag = 1;
                      for (int j = i; j < v_u->degree.out - 1; j++) {
00166
00167
                           v_u->adj_list[j] = v_u->adj_list[j + 1];
00168
00169
                      v_u->degree.out--;
00170
                 }
00171
            if (q->type == UNDIGRAPH && flag) {
00172
00173
                 for (int i = 0; i < v_v->degree.out; i++) {
                      <u>if</u>(v_v->adj_list[i] == u) {
00175
                           for (int j = i; j < v_v->degree.out - 1; j++) {
00176
                               v_v-adj_list[j] = v_v-adj_list[j + 1];
00177
00178
                           v v->degree.out--;
00179
                      }
00180
                 }
00181
00182
             if(flag) g->m--;
00183 }
00184
00192 void graph_dot_output(graph_t* q, char* filename) {
00193
           FILE* pfile;
                    = fopen(filename, "w");
00194
            pfile
00195
            if (pfile == NULL) perror ("Error opening file");
00196
            if (g->type == DIGRAPH) {
                 fprintf(pfile, "digraph g {\nnode [shape=\"circle\"];\n");
for(int i = 0; i < g->n; i++) {
    if(g->vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
    for(int j = 0; j < g->vertex[i]->degree.out; j++) {
        fprintf(pfile, "%d -> %d;\n", i, g->vertex[i]->adj_list[j]);
}
00197
00198
00199
00200
00201
00202
00203
                }
            } else if (g->type == UNDIGRAPH) {
   fprintf(pfile, "graph g {\nnode [shape=\"circle\"];\n");
   for(int i = 0; i < g->n; i++) {
00204
00205
00206
00207
                      if(g->vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
00208
                      for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
                          if(i <= g->vertex[i]->adj_list[j]) {
   fprintf(pfile, "%d -- %d;\n", i, g->vertex[i]->
00209
00210
       adj_list[j]);
00211
00212
                      }
00213
                }
00214
00215
            fprintf(pfile, "\n");
00216
            fclose(pfile):
00217
            //system(strcat(strcat(strcat(strcat("dot ", filename),"-Tps -o "), filename),".ps"));
00218 }
00219
00220 info_t* bfs(graph_t* g, int src) {
            int* color = malloc(g->n * sizeof(int));
int* pred = malloc(g->n * sizeof(int));
int* dist = malloc(g->n * sizeof(int));
00221
00222
00223
            assert(color && pred && dist);
00224
00225
            for (int i = 0; i < g -> n; i++) {
00226
                color[i] = 0;
                 pred[i] = -1;
dist[i] = g->n + 1;
00227
00228
00229
            }
```

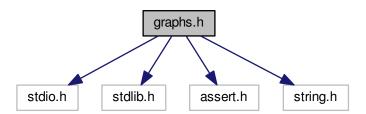
```
00230
           queue_t* q = queue_create(sizeof(int), g->n);
00231
           assert (q);
           assert(q),
int i = src;
for (int j = 0; j < g->n; j++) {
    if (color[i] == 0) {
      color[i] = 1;
}
00232
00233
00234
00235
                    dist[0];
pred[i] = -1;
00236
00237
00238
                     queue_enqueue(q, &i);
                    00239
00240
00241
00242
00243
00244
00245
00246
                                   queue_enqueue(q, &k);
00247
                              }
00248
                         queue_dequeue(q);
color[u] = 2;
00249
00250
00251
                     }
00252
                i = j;
00253
00254
00255
           free(color);
00256
            info_t* info;
           info->dist = dist;
info->pred = pred;
return info;
00257
00258
00259
00260 }
```

6.3 graphs.h File Reference

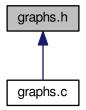
graph definition and basic operations

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
```

Include dependency graph for graphs.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct deg_t
- struct gvertex_t
- struct graph_t
- struct info_t

Enumerations

enum gtype_t { UNDIGRAPH, DIGRAPH }

Functions

- graph_t * graph_clone (graph_t *g)
- graph_t * graph_create (int n, gtype_t type)
- void graph_add_edge (graph_t *g, int u, int v)
- void graph_adj_list_print (graph_t *g)
- void graph_delete_adj_ele (graph_t *g, int u, int v)
- void graph_destruct (graph_t *g)
- void graph_dot_output (graph_t *g, char *filename)
- int is_adj (graph_t *g, int u, int v)
- info_t * bfs (graph_t *g, int src)

6.3.1 Detailed Description

graph definition and basic operations

Author

Firmin MARTIN

Version

0.1

Date

28/12/2017

Definition in file graphs.h.

6.3.2 Enumeration Type Documentation

6.3.2.1 enum gtype t

Enumerator

UNDIGRAPH undirect graphDIGRAPH direct graph

Definition at line 18 of file graphs.h.

```
00018 {
00019 UNDIGRAPH,
00020 DIGRAPH
00021 } gtype_t;
```

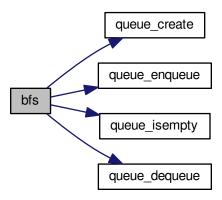
6.3.3 Function Documentation

```
6.3.3.1 info_t* bfs ( graph_t * g, int src )
```

Definition at line 220 of file graphs.c.

```
00220
00221
             int* color = malloc(g->n * sizeof(int));
             int* pred = malloc(g > n * sizeof(int));
int* dist = malloc(g - > n * sizeof(int));
00223
00224
             assert(color && pred && dist);
             for(int i = 0; i < g->n; i++) {
  color[i] = 0;
  pred[i] = -1;
  dist[i] = g->n + 1;
00225
00226
00227
00228
00229
00230
             queue_t* q = queue_create(sizeof(int), g->n);
00231
             assert(q);
             int i = src;
for (int j = 0; j < g->n; j++) {
    if (color[i] == 0) {
        color[i] = 1;
}
00232
00233
00234
00235
00236
                        dist[0];
                       pred[i] = -1;
00237
00238
                        queue_enqueue(q, &i);
00239
                        while(!queue_isempty(q)) {
                             int u = q->front;
for(int k = 0; k < g->vertex[u]->degree.out; k++) {
00240
00241
                                  if (color[k] == 0) {
    color[k] = 1;
00242
00243
                                        dist[k] = dist[u] + 1;
pred[k] = u;
00244
00245
                                        queue_enqueue(q, &k);
00246
00247
00248
                              queue_dequeue(q);
color[u] = 2;
00249
00250
00251
                        }
00252
00253
                  i = j;
00254
00255
             free(color);
00256
             info_t* info;
             info->dist = dist;
info->pred = pred;
return info;
00257
00258
00259
00260 }
```

Here is the call graph for this function:



6.3.3.2 void graph_add_edge (graph_t * g, int u, int v)

Add the edge (u, v) in the graph g. If g is an undigraph, (v, u) is also added.

Parameters

g	graph
и	vertex index
V	vertex index

Definition at line 78 of file graphs.c.

```
00078
         gvertex_t *v_u = g->vertex[u], *v_v = g->vertex[v];
assert(u >= 0 && v >= 0 && u < g->n && v < g->n);
00079
08000
         assert(v_u->degree.out <= v_u->adj_list_len);
00082
          if (is_adj(g, u, v)) return;
00083
         if (v_u->degree.out == v_u->adj_list_len) {
             int* newlist = realloc(v_u->adj_list, sizeof(int) * (v_u->
00084
     adj_list_len + 10));
00085
            assert(newlist);
             v_u->adj_list = newlist;
v_u->adj_list_len += 10;
00086
00087
00088
00089
         v_u->adj_list[v_u->degree.out] = v;
00090
         v_u->degree.out++;
         v_v->degree.in++;
00091
         if (g->type == UNDIGRAPH && u != v) {
00092
00093
              assert(v_v->degree.out <= v_v->adj_list_len);
00094
              if (v_v->degree.out == v_v->adj_list_len) {
                 00095
00096
                 assert(newlist);
00097
                 v_v->adj_list = newlist;
00098
                 v_v->adj_list_len += 10;
00099
00100
              v_v->adj_list[v_v->degree.out] = u;
00101
             v_v->degree.out++;
00102
             v_u->degree.in++;
00103
00104
         g->m++;
00105 }
```

Here is the call graph for this function:



```
6.3.3.3 void graph_adj_list_print ( graph_t * g )
```

Print graph's adjacency list representation

Parameters

```
g graph
```

Definition at line 111 of file graphs.c.

6.3.3.4 graph_t* graph_clone (graph_t * g)

Clone a graph

Parameters



Returns

graph cloned

Bug sizeof(pt) is wrong!!

Definition at line 131 of file graphs.c.

```
00131
00132
          graph_t* new_g;
00133
          new_g = (graph_t*) malloc(sizeof(graph_t));
00134
          if (new_g != NULL) {
00135
              new_g->type = g->type;
              new_g->n=g->n;
00136
              new_g->m = g->m;
00137
00138
              new_g->vertex = (gvertex_t**) malloc(sizeof(g->vertex));
00139
               assert(new_g->vertex);
              for(int i = 0; i < g->n; i++) {
   new_g->vertex[i] = (gvertex_t*) malloc(sizeof()
00140
00141
      gvertex_t));
00142
                  assert(q->vertex[i]);
                  new_g->vertex[i]->adj_list = (int*) malloc(sizeof(g->
      vertex[i]->adj_list));
00144
                  assert(new_g->vertex[i]->adj_list);
00145
                  new_g->vertex[i]->adj_list_len = g->vertex[i]->
      adj_list_len;
00146
                  new_q->vertex[i]->degree.in = q->vertex[i]->
      degree.in;
00147
                  new_g->vertex[i]->degree.out = g->vertex[i]->
      degree.out;
00148
00149
00150
          return new_q;
00151 }
```

6.3.3.5 graph t* graph_create (int n, gtype t type)

Create a graph initialized as a forest with n vertices

Parameters

n		number of vertices
typ	ре	type of graph (digraph, undigraph)

Returns

return a graph initialized as a forest

Definition at line 19 of file graphs.c.

```
00019
00020
           graph_t* g;
00021
           g = (graph_t*) malloc(sizeof(graph_t));
00022
           assert(g);
00023
           if (g != NULL) {
                g->type = type; /* initialize type of graph */
               g->n = n;  /* initialize number of vertices */
g->m = 0;  /* g is a forest => 0 edge */
00025
00026
00027
                g->vertex = (gvertex_t**) malloc(sizeof(gvertex_t*) * n);
00028
                assert(g->vertex);
               for(int i = 0; i < g->n; i++) {
    g->vertex[i] = (gvertex_t*) malloc(sizeof(gvertex_t));
00029
00030
00031
                    assert(g->vertex[i]);
00032
                    /* initialize an array with size 5 by default */
00033
                    g->vertex[i]->adj_list = (int*) malloc(sizeof(int) * 5);
00034
                    assert(g->vertex[i]->adj_list);
                    g->vertex(i)->adj_list_len = 5;
/* g is a forest => deg_in(i) = deg_out(i) = 0 */
00035
00036
00037
                    g->vertex[i]->degree.in = 0;
00038
                    g->vertex[i]->degree.out = 0;
00039
               }
00040
           return q;
00041
00042 }
```

6.3.3.6 void graph_delete_adj_ele (graph_t * g, int u, int v)

Delete the edge (u,v) in graph g. If g is an undigraph, (v,u) is also removed.

Parameters

g	graph
и	vertex index
V	vertex index

Definition at line 159 of file graphs.c.

```
00159
              gvertex_t *v_u = g->vertex[u], *v_v = g->vertex[v];
assert(v_u && v_v);
00160
00161
00162
               int flag = 0;
00163
               for (int i = 0; i < v_u->degree.out; i++) {
00164
                      if(v_u->adj_list[i] == v) {
00165
                           flag = 1;
                           for (int j = i; j < v_u->degree.out - 1; j++) {
    v_u->adj_list[j] = v_u->adj_list[j + 1];
00166
00167
00168
00169
                           v_u->degree.out--;
00170
00171
              if (g->type == UNDIGRAPH && flag) {
   for (int i = 0; i < v_v->degree.out; i++) {
      if(v_v->adj_list[i] == u) {
        for (int j = i; j < v_v->degree.out - 1; j++) {
            v_v->adj_list[j] = v_v->adj_list[j + 1];
      }
}
00172
00173
00174
00175
00176
00177
00178
                                  v_v->degree.out--;
00179
                           }
00180
                     }
00181
00182
               if(flag) g->m--;
00183 }
```

6.3.3.7 void graph_destruct (graph_t * g)

Free a graph

Parameters

```
g a graph
```

Definition at line 48 of file graphs.c.

6.3.3.8 void graph_dot_output (graph_t * g, char * filename)

Output the graph g in dot format (filename.dot) and use dot compile it to filename.ps .

Parameters

g	graph
filename	filename without any extension

6.4 graphs.h 25

Todo system call?

Definition at line 192 of file graphs.c.

```
00192
              FILE* pfile;
pfile = fopen(filename, "w");
00193
00194
00195
               if (pfile == NULL) perror ("Error opening file");
00196
               if (g->type == DIGRAPH) {
                    fprintf(pfile, "digraph g {\nnode [shape=\"circle\"];\n");
for(int i = 0; i < g->n; i++) {
00197
00198
                          fif (g>vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
for(int j = 0; j < g->vertex[i]->degree.out; j++) {
    fprintf(pfile, "%d -> %d;\n", i, g->vertex[i]->adj_list[j]);
00199
00200
00201
00202
00203
                    }
              } else if (g->type == UNDIGRAPH) {
  fprintf(pfile, "graph g {\nnode [shape=\"circle\"];\n");
  for(int i = 0; i < g->n; i++) {
00204
00205
00206
                          if(g->vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
                          for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
    if(i <= g->vertex[i]->adj_list[j]) {
        fprintf(pfile, "%d -- %d;\n", i, g->vertex[i]->
00208
00209
00210
        adj_list[j]);
00211
00212
00213
                    }
00214
00215
              fprintf(pfile, "\n");
00216
              fclose(pfile);
00217
              //system(strcat(strcat(strcat(strcat("dot ", filename),"-Tps -o "), filename),".ps"));
00218 }
```

6.3.3.9 int is_adj (graph_t * g, int u, int v)

Determinate if v is in the adjacency list of u in graph g

Parameters

g	graph
и	vertex index
V	vertex index

Definition at line 63 of file graphs.c.

6.4 graphs.h

```
00001 #ifndef GRAPHS_H
00002 #define GRAPHS_H
00003
00012 #include <stdio.h>
00013 #include <stdlib.h>
00014 #include <assert.h>
00015 #include <string.h>
00016
00018 typedef enum {
00019 UNDIGRAPH,
```

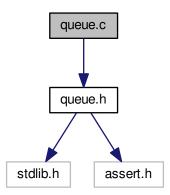
```
00020
               DIGRAPH
00021 } gtype_t;
00022
00023 typedef struct deg_t {
00024
             int in;
int out;
00025
00026 } deg_t;
00027
00028 typedef struct gvertex_t {
              int* adj_list;
int adj_list_len;
deg_t degree;
00029
00030
00031
00032 } gvertex_t;
00034 typedef struct graph_t {
             gtype_t type;
int n;
int m;
00035
00036
00037
               gvertex_t** vertex;
00039 } graph_t;
00040
00041 typedef struct info_t {
00042
          int* pred;
int* dist;
00043
00044 } info_t;
00046 graph_t* graph_clone(graph_t* g);
00047 graph_t* graph_create(int n, gtype_t type);
00048 void graph_add_edge(graph_t* g, int u, int v);
00049 void graph_adj_list_print(graph_t* g);
00050 void graph_delete_adj_ele(graph_t* g, int u, int v);
00051 void graph_destruct(graph_t* g);
00052 void graph_dot_output(graph_t* g, char* filename);
00053 int is_adj(graph_t* g, int u, int v);
00054 info_t* bfs(graph_t* g, int src);
00055
00056 #endif //GRAPHS_H
```

6.5 queue.c File Reference

queue's basic operations implementation (using dynamic array)

```
#include "queue.h"
```

Include dependency graph for queue.c:



Functions

int queue_isempty (queue_t *q)

- void queue_enqueue (queue_t *q, void *e)
- void * queue_dequeue (queue_t *q)
- queue_t * queue_create (size_t width, int max_size)
- void queue_destruct (queue_t *q)

6.5.1 Detailed Description

queue's basic operations implementation (using dynamic array)

Author

Firmin MARTIN

Version

0.1

Date

28/12/2017

Definition in file queue.c.

6.5.2 Function Documentation

```
6.5.2.1 queue_t* queue_create ( size_t width, int max_size )
```

Given the size of each element and the queue size, create a queue.

Parameters

width	size of each element
max_size	size of the queue, max_size*width bytes will be reserved (definitively) for the queue

Returns

a queue initialized

Note

This queue implementation assume that the amount of element will never exceed max_size. See queue_ cenqueue for more information on the behavior in the excess case.

Definition at line 60 of file queue.c.

6.5.2.2 void* queue_dequeue (queue_t * q)

Dequeue an element from the queue s.

Parameters

```
q queue
```

Returns

an element

Definition at line 43 of file queue.c.

6.5.2.3 void queue_destruct (queue_t * q)

Free a queue.

Parameters

```
q a queue
```

Definition at line 77 of file queue.c.

```
00077

00078 free(q->base);

00079 free(q);

00080 }
```

6.5.2.4 void queue_enqueue (queue_t * q, void * e)

Enqueue an element e into the queue q.

Parameters

q	queue
е	element which be enqueued

6.6 queue.c 29

Note

Note that if the max size is reached, this function will overwrite the queue and consider the queue as empty, i.e. at the end the queue has just one element.

Definition at line 30 of file queue.c.

6.5.2.5 int queue_isempty (queue_t * q)

Determinate the emptiness of a queue.

Parameters

```
s queue
```

Returns

1 if the queue s is empty, 0 otherwise.

Definition at line 17 of file queue.c.

6.6 queue.c

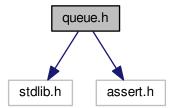
```
00001
00009 #include "queue.h"
00010
00017 int queue_isempty(queue_t* q) {
00018
          return q->count == 0;
00019 }
00020
00030 void queue_enqueue(queue_t* q, void* e) {
        q->base[q->front] = e;
if (q->front == q->max_size - 1) q->front = 0;
00031
00032
          else q->front++;
00034
          q->count = (q->count + 1) % q->max_size;
00035 }
00036
00043 void* queue_dequeue(queue_t* q) {
00044 void* e = q->base[(q->front - q->count + q->max_size)%q->
      max_size];
        q->count-
00045
00046
          return e;
00047 }
00048
00060 queue_t* queue_create(size_t width, int max_size) {
          queue_t* q = malloc(sizeof(queue_t));
00061
00062
          assert (q);
00063
           q->width = width;
00064
           q->max_size = max_size ;
00065
          q->base = (void**) calloc(q->max_size, sizeof(void*));
          assert (q->base);
00066
00067
          q \rightarrow front = 0;
00068
          q->count = 0;
```

6.7 queue.h File Reference

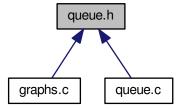
queue (using array) definition and basic operations

```
#include <stdlib.h>
#include <assert.h>
```

Include dependency graph for queue.h:



This graph shows which files directly or indirectly include this file:



Data Structures

struct queue_t

Functions

- queue_t * queue_create (size_t width, int max_size)
- void queue_destruct (queue_t *q)
- int queue_isempty (queue_t *q)
- void * queue_dequeue (queue_t *q)
- void queue_enqueue (queue_t *q, void *e)

6.7.1 Detailed Description

queue (using array) definition and basic operations

Author

Firmin MARTIN

Version

0.1

Date

28/12/2017

Definition in file queue.h.

6.7.2 Function Documentation

6.7.2.1 queue_t* queue_create (size_t width, int max_size)

Given the size of each element and the queue size, create a queue.

Parameters

width	size of each element
max_size	size of the queue, max_size*width bytes will be reserved (definitively) for the queue

Returns

a queue initialized

Note

This queue implementation assume that the amount of element will never exceed max_size. See queue_
enqueue for more information on the behavior in the excess case.

Definition at line 60 of file queue.c.

```
00061
            queue_t* q = malloc(sizeof(queue_t));
00062
            assert(q);
00063
            q->width = width;
00064
            q->max_size = max_size;
q->base = (void**) calloc(q->max_size, sizeof(void*));
assert(q->base);
00065
00066
            q->front = 0;
q->count = 0;
00067
00068
00069
            return q;
00070 }
```

6.7.2.2 void* queue_dequeue (queue_t * q)

Dequeue an element from the queue s.

Parameters

```
q queue
```

Returns

an element

Definition at line 43 of file queue.c.

6.7.2.3 void queue_destruct (queue_t * q)

Free a queue.

Parameters

```
q a queue
```

Definition at line 77 of file queue.c.

```
00077
00078 free(q->base);
00079 free(q);
```

6.7.2.4 void queue_enqueue (queue_t * q, void * e)

Enqueue an element e into the queue q.

Parameters

q	queue
е	element which be enqueued

6.8 queue.h 33

Note

Note that if the max size is reached, this function will overwrite the queue and consider the queue as empty, i.e. at the end the queue has just one element.

Definition at line 30 of file queue.c.

6.7.2.5 int queue_isempty (queue_t * q)

Determinate the emptiness of a queue.

Parameters

```
s queue
```

Returns

1 if the queue s is empty, 0 otherwise.

Definition at line 17 of file queue.c.

6.8 queue.h

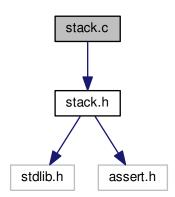
```
00001 #ifndef STACK_H
00002 #define STACK_H
00003
00004 #include <stdlib.h>
00005 #include <assert.h>
00006
00021 typedef struct {
       size_t width;
00022
00023
         int front;
00024
         int count;
        void** base;
int max_size;
00025
00026
00027 } queue_t;
00029 queue_t* queue_create(size_t width, int max_size);
00030 void queue_destruct(queue_t* q);
00031 int queue_isempty(queue_t* q);
00032 void* queue_dequeue(queue_t* q);
00033 void queue_enqueue(queue_t* q, void* e);
00034
00035 #endif /* ifndef STACK_H */
```

6.9 stack.c File Reference

stack's basic operations implementation (using dynamic array)

```
#include "stack.h"
```

Include dependency graph for stack.c:



Functions

- int stack_isempty (stack_t *s)
- void stack_push (stack_t *s, void *e)
- void * stack_pop (stack_t *s)
- stack_t * stack_create (size_t width)
- void stack_destruct (stack_t *s)

6.9.1 Detailed Description

stack's basic operations implementation (using dynamic array)

Author

Firmin MARTIN

Version

0.1

Date

28/12/2017

Definition in file stack.c.

6.9.2 Function Documentation

6.9.2.1 stack_t* stack_create (size_t width)

Given the size of each element, create a stack 10 * sizeof(void*) bytes is reserved by default.

Parameters

width size of each ele	ement
------------------------	-------

Returns

a stack initialized

Definition at line 57 of file stack.c.

6.9.2.2 void stack_destruct (stack_t * s)

Free a stack.

Parameters

```
s a stack
```

Definition at line 73 of file stack.c.

6.9.2.3 int stack_isempty ($stack_t * s$)

Determinate the emptiness of a stack.

Parameters

```
s stack
```

Returns

1 if the stack s is empty, 0 otherwise.

Definition at line 17 of file stack.c.

```
6.9.2.4 void* stack_pop ( stack_t * s )
```

Pop out an element from the stack s.

Parameters

```
s stack
```

Returns

an element

Definition at line 44 of file stack.c.

```
00044 {
00045 if (stack_isempty(s)) return NULL;
00046 s->top--;
00047 return s->base[s->top + 1];
00048 }
```

Here is the call graph for this function:



```
6.9.2.5 void stack_push ( stack_t * s, void * e )
```

Push an element e into the stack s.

Parameters

s	stack
е	element which be pushed

Definition at line 27 of file stack.c.

6.10 stack.c 37

6.10 stack.c

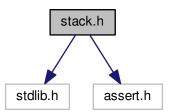
```
00001
00009 #include "stack.h"
00010
00017 int stack_isempty(stack_t* s) {
00018
         return s->top == -1;
00019 }
00020
00027 void stack_push(stack_t* s, void* e) {
        s->top++;
00028
          if (s->top == s->mem_size) {
00029
00030
             void** newptr = realloc(s->base, sizeof(void*) * (s->mem_size + 10));
00031
             assert(newptr);
00032
             s->base = newptr;
             s->mem_size += 10;
00033
00034
00035
         s->base[s->top] = e;
00036 }
00037
00044 void* stack_pop(stack_t* s) {
00045
        if (stack_isempty(s)) return NULL;
00046
         s->top--;
00047
         return s->base[s->top + 1];
00048 }
00049
00057 stack_t* stack_create(size_t width) {
       stack_t* s = malloc(sizeof(stack_t));
00058
00059
         assert(s);
         s->width = width;
00060
00061
         s->mem_size = 10;
00062
         s->base = (void**) malloc(sizeof(void*) * s->mem_size);
00063
         assert(s->base);
00064
         s->top = -1;
00065
         return s:
00066 }
00067
00073 void stack_destruct(stack_t* s) {
00074
        free(s->base);
00075
         free(s);
00076 }
00077
00078
00079
```

6.11 stack.h File Reference

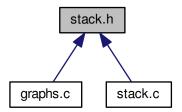
stack definition and basic operations

```
#include <stdlib.h>
#include <assert.h>
```

Include dependency graph for stack.h:



This graph shows which files directly or indirectly include this file:



Data Structures

struct stack_t

Functions

- stack_t * stack_create (size_t width)
- void stack_destruct (stack_t *s)
- int stack_isempty (stack_t *s)
- void * stack_pop (stack_t *s)
- void stack_push (stack_t *s, void *e)

6.11.1 Detailed Description

stack definition and basic operations

Author

Firmin MARTIN

Version

0.1

Date

28/12/2017

Definition in file stack.h.

6.11.2 Function Documentation

6.11.2.1 stack_t* stack_create (size_t width)

Given the size of each element, create a stack 10 * sizeof(void*) bytes is reserved by default.

Parameters

width size of each ele	ement
------------------------	-------

Returns

a stack initialized

Definition at line 57 of file stack.c.

6.11.2.2 void stack_destruct (stack_t * s)

Free a stack.

Parameters

```
s a stack
```

Definition at line 73 of file stack.c.

6.11.2.3 int stack_isempty ($stack_t * s$)

Determinate the emptiness of a stack.

Parameters

```
s stack
```

Returns

1 if the stack s is empty, 0 otherwise.

Definition at line 17 of file stack.c.

```
6.11.2.4 void* stack_pop ( stack_t * s )
```

Pop out an element from the stack s.

Parameters

```
s stack
```

Returns

an element

Definition at line 44 of file stack.c.

```
00044 {
00045 if (stack_isempty(s)) return NULL;
00046 s->top--;
00047 return s->base[s->top + 1];
00048 }
```

Here is the call graph for this function:



```
6.11.2.5 void stack_push ( stack_t * s, void * e )
```

Push an element e into the stack s.

Parameters

s	stack
е	element which be pushed

Definition at line 27 of file stack.c.

6.12 stack.h 41

6.12 stack.h

Index

adj_list	graph_add_edge, 11
gvertex_t, 6	graph_adj_list_print, 12
adj list len	graph_clone, 12
gvertex_t, 6	graph_create, 13
9.0.0, 0	graph_delete_adj_ele, 13
base	graph_destruct, 14
queue_t, 7	graph_dot_output, 14
• —	
stack_t, 8	is_adj, 15
bfs	graphs.h, 18
graphs.c, 10	bfs, 20
graphs.h, 20	DIGRAPH, 20
	graph_add_edge, 21
count	graph_adj_list_print, 22
queue_t, 7	graph_clone, 22
	graph_create, 23
DIGRAPH	graph_delete_adj_ele, 23
graphs.h, 20	graph destruct, 24
deg_t, 3	graph_dot_output, 24
in, 3	gtype_t, 20
out, 3	is_adj, 25
degree	UNDIGRAPH, 20
gvertex_t, 6	
dist	gtype_t
info_t, 6	graphs.h, 20
1110_t, 0	gvertex_t, 5
front	adj_list, 6
queue_t, 7	adj_list_len, 6
queue_t, /	degree, 6
graph_add_edge	
	in
graphs.c, 11	deg_t, 3
graphs.c, 11 graphs.h, 21	deg_t, 3 info_t, 6
graphs.c, 11 graphs.h, 21 graph_adj_list_print	deg_t, 3 info_t, 6 dist, 6
graphs.c, 11 graphs.h, 21 graph_adj_list_print graphs.c, 12	deg_t, 3 info_t, 6
graphs.c, 11 graphs.h, 21 graph_adj_list_print graphs.c, 12 graphs.h, 22	deg_t, 3 info_t, 6 dist, 6
graphs.c, 11 graphs.h, 21 graph_adj_list_print graphs.c, 12 graphs.h, 22 graph_clone	deg_t, 3 info_t, 6 dist, 6 pred, 6
graphs.c, 11 graphs.h, 21 graph_adj_list_print graphs.c, 12 graphs.h, 22 graph_clone graphs.c, 12	deg_t, 3 info_t, 6 dist, 6 pred, 6 is_adj
graphs.c, 11 graphs.h, 21 graph_adj_list_print graphs.c, 12 graphs.h, 22 graph_clone	deg_t, 3 info_t, 6 dist, 6 pred, 6 is_adj graphs.c, 15
graphs.c, 11 graphs.h, 21 graph_adj_list_print graphs.c, 12 graphs.h, 22 graph_clone graphs.c, 12	deg_t, 3 info_t, 6 dist, 6 pred, 6 is_adj graphs.c, 15
graphs.c, 11 graphs.h, 21 graph_adj_list_print graphs.c, 12 graphs.h, 22 graph_clone graphs.c, 12 graphs.c, 12 graphs.h, 22	deg_t, 3 info_t, 6 dist, 6 pred, 6 is_adj graphs.c, 15 graphs.h, 25
graphs.c, 11 graphs.h, 21 graph_adj_list_print graphs.c, 12 graphs.h, 22 graph_clone graphs.c, 12 graphs.h, 22 graph_create	deg_t, 3 info_t, 6 dist, 6 pred, 6 is_adj graphs.c, 15 graphs.h, 25
graphs.c, 11 graphs.h, 21 graph_adj_list_print graphs.c, 12 graphs.h, 22 graph_clone graphs.c, 12 graphs.h, 22 graph_create graphs.c, 13 graphs.h, 23	deg_t, 3 info_t, 6 dist, 6 pred, 6 is_adj graphs.c, 15 graphs.h, 25 m graph_t, 4
graphs.c, 11 graphs.h, 21 graph_adj_list_print graphs.c, 12 graphs.h, 22 graph_clone graphs.c, 12 graphs.h, 22 graph_create graphs.c, 13 graphs.h, 23 graph_delete_adj_ele	deg_t, 3 info_t, 6 dist, 6 pred, 6 is_adj graphs.c, 15 graphs.h, 25 m graph_t, 4 max_size queue_t, 7
graphs.c, 11 graphs.h, 21 graph_adj_list_print graphs.c, 12 graphs.h, 22 graph_clone graphs.c, 12 graphs.h, 22 graph_create graphs.c, 13 graphs.h, 23 graph_delete_adj_ele graphs.c, 13	deg_t, 3 info_t, 6 dist, 6 pred, 6 is_adj graphs.c, 15 graphs.h, 25 m graph_t, 4 max_size queue_t, 7 mem_size
graphs.c, 11 graphs.h, 21 graph_adj_list_print graphs.c, 12 graphs.h, 22 graph_clone graphs.c, 12 graphs.h, 22 graph_create graph_create graphs.c, 13 graphs.h, 23 graph_delete_adj_ele graphs.c, 13 graphs.h, 23	deg_t, 3 info_t, 6 dist, 6 pred, 6 is_adj graphs.c, 15 graphs.h, 25 m graph_t, 4 max_size queue_t, 7
graphs.c, 11 graphs.h, 21 graph_adj_list_print graphs.c, 12 graphs.h, 22 graph_clone graphs.c, 12 graphs.h, 22 graph_create graphs.c, 13 graphs.h, 23 graph_delete_adj_ele graphs.c, 13 graphs.h, 23 graph_destruct	deg_t, 3 info_t, 6 dist, 6 pred, 6 is_adj graphs.c, 15 graphs.h, 25 m graph_t, 4 max_size queue_t, 7 mem_size
graphs.c, 11 graphs.h, 21 graph_adj_list_print graphs.c, 12 graphs.h, 22 graph_clone graphs.c, 12 graphs.h, 22 graph_create graphs.c, 13 graphs.h, 23 graph_delete_adj_ele graphs.c, 13 graphs.h, 23 graph_destruct graphs.c, 14	deg_t, 3 info_t, 6 dist, 6 pred, 6 is_adj graphs.c, 15 graphs.h, 25 m graph_t, 4 max_size queue_t, 7 mem_size stack_t, 8 n
graphs.c, 11 graphs.h, 21 graph_adj_list_print graphs.c, 12 graphs.h, 22 graph_clone graphs.c, 12 graphs.h, 22 graph_create graphs.c, 13 graphs.h, 23 graph_delete_adj_ele graphs.c, 13 graphs.h, 23 graph_destruct graphs.c, 14 graphs.h, 24	deg_t, 3 info_t, 6 dist, 6 pred, 6 is_adj graphs.c, 15 graphs.h, 25 m graph_t, 4 max_size queue_t, 7 mem_size stack_t, 8
graphs.c, 11 graphs.h, 21 graph_adj_list_print graphs.c, 12 graphs.h, 22 graph_clone graphs.c, 12 graphs.h, 22 graph_create graphs.c, 13 graphs.h, 23 graph_delete_adj_ele graphs.c, 13 graphs.h, 23 graph_destruct graphs.c, 14 graphs.h, 24 graph_dot_output	deg_t, 3 info_t, 6 dist, 6 pred, 6 is_adj graphs.c, 15 graphs.h, 25 m graph_t, 4 max_size queue_t, 7 mem_size stack_t, 8 n
graphs.c, 11 graphs.h, 21 graph_adj_list_print graphs.c, 12 graphs.h, 22 graph_clone graphs.c, 12 graphs.h, 22 graph_create graphs.c, 13 graphs.h, 23 graph_delete_adj_ele graphs.c, 13 graphs.h, 23 graph_destruct graphs.c, 14 graphs.h, 24 graph_dot_output graphs.c, 14	deg_t, 3 info_t, 6 dist, 6 pred, 6 is_adj graphs.c, 15 graphs.h, 25 m graph_t, 4 max_size queue_t, 7 mem_size stack_t, 8 n graph_t, 4
graphs.c, 11 graphs.h, 21 graph_adj_list_print graphs.c, 12 graphs.h, 22 graph_clone graphs.c, 12 graphs.h, 22 graph_create graphs.c, 13 graphs.h, 23 graph_delete_adj_ele graphs.c, 13 graphs.h, 23 graph_destruct graphs.h, 23 graph_destruct graphs.c, 14 graphs.h, 24 graph_dot_output graphs.c, 14 graphs.h, 24	deg_t, 3 info_t, 6 dist, 6 pred, 6 is_adj graphs.c, 15 graphs.h, 25 m graph_t, 4 max_size queue_t, 7 mem_size stack_t, 8 n graph_t, 4 out
graphs.c, 11 graphs.h, 21 graph_adj_list_print graphs.c, 12 graphs.h, 22 graph_clone graphs.c, 12 graphs.h, 22 graph_create graphs.c, 13 graphs.h, 23 graph_delete_adj_ele graphs.c, 13 graphs.h, 23 graph_destruct graphs.c, 14 graphs.h, 24 graph_dot_output graphs.c, 14 graphs.h, 24 graph_t, 4	deg_t, 3 info_t, 6 dist, 6 pred, 6 is_adj graphs.c, 15 graphs.h, 25 m graph_t, 4 max_size queue_t, 7 mem_size stack_t, 8 n graph_t, 4 out
graphs.c, 11 graphs.h, 21 graph_adj_list_print graphs.c, 12 graphs.h, 22 graph_clone graphs.c, 12 graphs.h, 22 graph_create graphs.c, 13 graphs.h, 23 graph_delete_adj_ele graphs.c, 13 graphs.h, 23 graph_destruct graphs.c, 14 graphs.h, 24 graph_dot_output graphs.c, 14 graphs.h, 24 graph_t, 4 m, 4	deg_t, 3 info_t, 6 dist, 6 pred, 6 is_adj graphs.c, 15 graphs.h, 25 m graph_t, 4 max_size queue_t, 7 mem_size stack_t, 8 n graph_t, 4 out deg_t, 3
graphs.c, 11 graphs.h, 21 graph_adj_list_print graphs.c, 12 graphs.h, 22 graph_clone graphs.c, 12 graphs.h, 22 graph_create graphs.c, 13 graphs.h, 23 graph_delete_adj_ele graphs.c, 13 graphs.h, 23 graph_destruct graphs.c, 14 graphs.h, 24 graph_dot_output graphs.c, 14 graphs.h, 24 graph_t, 4 m, 4 n, 4	deg_t, 3 info_t, 6 dist, 6 pred, 6 is_adj graphs.c, 15 graphs.h, 25 m graph_t, 4 max_size queue_t, 7 mem_size stack_t, 8 n graph_t, 4 out deg_t, 3 pred
graphs.c, 11 graph_adj_list_print graphs.c, 12 graphs.h, 22 graph_clone graphs.c, 12 graphs.h, 22 graph_create graphs.c, 13 graphs.h, 23 graph_delete_adj_ele graphs.c, 13 graphs.h, 23 graph_destruct graphs.c, 14 graphs.h, 24 graph_dot_output graphs.c, 14 graphs.h, 24 graph_t, 4 m, 4 n, 4 type, 4	deg_t, 3 info_t, 6 dist, 6 pred, 6 is_adj graphs.c, 15 graphs.h, 25 m graph_t, 4 max_size queue_t, 7 mem_size stack_t, 8 n graph_t, 4 out deg_t, 3 pred info_t, 6 queue.c, 26
graphs.c, 11 graphs.h, 21 graph_adj_list_print graphs.c, 12 graphs.h, 22 graph_clone graphs.c, 12 graphs.h, 22 graph_create graphs.c, 13 graphs.h, 23 graph_delete_adj_ele graphs.c, 13 graphs.h, 23 graph_destruct graphs.c, 14 graphs.h, 24 graph_dot_output graphs.c, 14 graphs.h, 24 graph_t, 4 m, 4 n, 4	deg_t, 3 info_t, 6 dist, 6 pred, 6 is_adj graphs.c, 15 graphs.h, 25 m graph_t, 4 max_size queue_t, 7 mem_size stack_t, 8 n graph_t, 4 out deg_t, 3 pred info_t, 6
graphs.c, 11 graph_adj_list_print graphs.c, 12 graphs.h, 22 graph_clone graphs.c, 12 graphs.h, 22 graph_create graphs.c, 13 graphs.h, 23 graph_delete_adj_ele graphs.c, 13 graphs.h, 23 graph_destruct graphs.c, 14 graphs.h, 24 graph_dot_output graphs.c, 14 graphs.h, 24 graph_t, 4 m, 4 n, 4 type, 4	deg_t, 3 info_t, 6 dist, 6 pred, 6 is_adj graphs.c, 15 graphs.h, 25 m graph_t, 4 max_size queue_t, 7 mem_size stack_t, 8 n graph_t, 4 out deg_t, 3 pred info_t, 6 queue.c, 26
graphs.c, 11 graphs.h, 21 graph_adj_list_print graphs.c, 12 graphs.h, 22 graph_clone graphs.c, 12 graphs.h, 22 graph_create graphs.c, 13 graphs.h, 23 graph_delete_adj_ele graphs.c, 13 graphs.h, 23 graph_destruct graphs.c, 14 graphs.h, 24 graph_dot_output graphs.c, 14 graphs.h, 24 graph_t, 4 m, 4 n, 4 type, 4 vertex, 5	deg_t, 3 info_t, 6 dist, 6 pred, 6 is_adj graphs.c, 15 graphs.h, 25 m graph_t, 4 max_size queue_t, 7 mem_size stack_t, 8 n graph_t, 4 out deg_t, 3 pred info_t, 6 queue_create, 27

44 INDEX

queue_enqueue, 28 queue_isempty, 29 queue.h, 30	t	mem_size, 8 op, 8 vidth, 8
queue_create, 31	4	
queue_dequeue, 32	top	stack t, 8
queue_destruct, 32	type	stack_t, o
queue_enqueue, 32 queue_isempty, 33		graph_t, 4
queue_create		, -1- =-
queue.c, 27		GRAPH
queue.h, 31	Ç	graphs.h, <mark>20</mark>
queue_dequeue	vertex	,
queue.c, 28		yraph_t, 5
queue.h, 32		,. «pt, •
queue_destruct	width	
queue.c, 28 queue.h, 32		queue_t, 7
queue_enqueue	8	stack_t, 8
queue.c, 28		
queue.h, 32		
queue_isempty		
queue.c, 29		
queue.h, 33		
queue_t, 7		
base, 7		
count, 7		
front, 7 max_size, 7		
width, 7		
stack.c, 34		
stack_create, 34		
stack_destruct, 35		
stack_isempty, 35		
stack_pop, 35 stack_push, 36		
stack.h, 37		
stack_create, 38		
stack_destruct, 39		
stack_isempty, 39		
stack_pop, 39		
stack_push, 40		
stack_create		
stack.c, 34		
stack.h, 38 stack destruct		
stack.c, 35		
stack.h, 39		
stack_isempty		
stack.c, 35		
stack.h, 39		
stack_pop		
stack.c, 35		
stack.h, 39		
ata ak muah		
stack_push		
stack.c, 36		
		