Graph

Generated by Doxygen 1.8.11

Contents

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:

4 File Index

4.1 File List

Here is a list of all files with brief descriptions:

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.

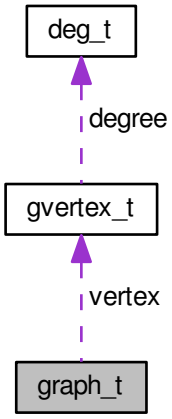
The documentation for this struct was generated from the following file:

- 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.

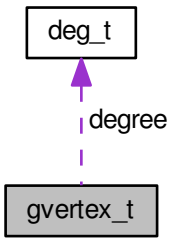
The documentation for this struct was generated from the following file:

- 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.

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

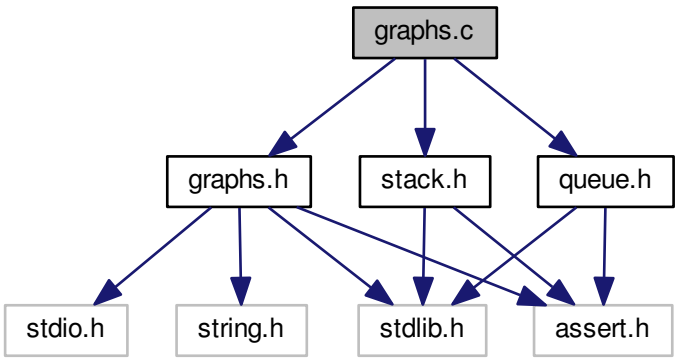
6 File Documentation

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.

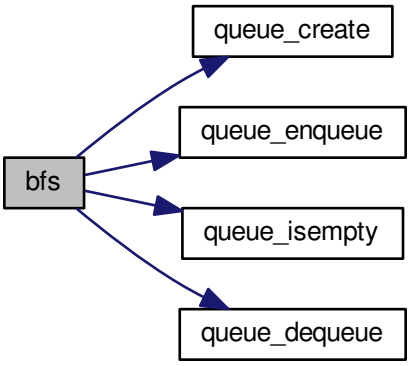
6.1.2 Function Documentation

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

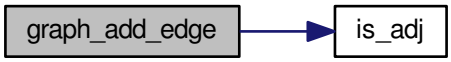
Parameters

g	graph
u	vertex index
v	vertex index

Definition at line 78 of file graphs.c.

```
00078                                           {
00079     gvertex_t *v_u = g->vertex[u], *v_v = g->vertex[v];
00080     assert(u >= 0 && v >= 0 && u < g->n && v < g->n);
00081     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) {
00084         int* newlist = realloc(v_u->adj_list, sizeof(int) * (v_u->
      adj_list_len + 10));
00085         assert(newlist);
00086         v_u->adj_list = newlist;
00087         v_u->adj_list_len += 10;
00088     }
00089     v_u->adj_list[v_u->degree.out] = v;
00090     v_u->degree.out++;
00091     v_v->degree.in++;
00092     if (g->type == UNDIGRAPH && u != v) {
00093         assert(v_v->degree.out <= v_v->adj_list_len);
00094         if (v_v->degree.out == v_v->adj_list_len) {
00095             int* newlist = realloc(v_v->adj_list, sizeof(int) * (v_v->adj_list_len + 10));
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.

```
00111                                       {
00112     if (g->type == DIGRAPH) printf("type : digraph\n");
00113     else if (g->type == UNDIGRAPH) printf("type : undigraph\n");
00114     printf("n=%d, m=%d\n", g->n, g->m);
00115     for(int i = 0; i < g->n; i++) {
00116         printf("[%d] ", i);
00117         if (g->vertex[i]->degree.out > 0) printf("-> ");
00118         for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
00119             printf("[%d]", g->vertex[i]->adj_list[j]);
00120         }
00121         printf("\n");
00122     }
00123 }
```

6.1.2.4 graph_t* graph_clone (graph_t * g)

Clone a graph

Parameters

g	graph

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;
00136         new_g->n = g->n;
00137         new_g->m = g->m;
00138         new_g->vertex = (gvertex_t**) malloc(sizeof(g->vertex));
00139         assert(new_g->vertex);
00140         for(int i = 0; i < g->n ; i++) {
00141             new_g->vertex[i] = (gvertex_t*) malloc(sizeof(
      gvertex_t));
00142             assert(g->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);
00145             new_g->vertex[i]->adj_list_len = g->vertex[i]->
      adj_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 }
```

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) {
00024         g->type = type;  /* initialize type of graph */
00025         g->n = n;        /* initialize number of vertices */
00026         g->m = 0;        /* g is a forest => 0 edge */
00027         g->vertex = (gvertex_t**) malloc(sizeof(gvertex_t*) * n);
00028         assert(g->vertex);
00029         for(int i = 0; i < g->n ; i++) {
00030             g->vertex[i] = (gvertex_t*) malloc(sizeof(gvertex_t));
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);
00035             g->vertex[i]->adj_list_len = 5;
00036             /* g is a forest => deg_in(i) = deg_out(i) = 0 */
00037             g->vertex[i]->degree.in = 0;
00038             g->vertex[i]->degree.out = 0;
00039         }
00040     }
00041     return g;
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
u	vertex index
v	vertex index

Definition at line 159 of file graphs.c.

```
00159                                                       {
00160     gvertex_t *v_u = g->vertex[u], *v_v = g->vertex[v];
00161     assert(v_u && v_v);
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;
00166             for (int j = i; j < v_u->degree.out - 1; j++) {
00167                 v_u->adj_list[j] = v_u->adj_list[j + 1];
00168             }
00169             v_u->degree.out--;
00170         }
00171     }
00172     if (g->type == UNDIGRAPH && flag) {
00173         for (int i = 0; i < v_v->degree.out; i++) {
00174             if(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 }
```

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.

```
00048                                   {
00049     for (int i = 0; i < g->n ; i++) {
00050         free(g->vertex[i]->adj_list);
00051         free(g->vertex[i]);
00052     }
00053     free(g->vertex);
00054     free(g);
00055 }
```

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

Todo system call ?

Definition at line 192 of file graphs.c.

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

6.1.2.9 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
u	vertex index
v	vertex index

Definition at line 63 of file graphs.c.

```
00063                                       {
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 }
```

6.2 graphs.c

```
00001
00009 #include "graphs.h"
00010 #include "queue.h"
00011 #include "stack.h"
00012
00019 graph_t* graph_create(int n, gtype_t type) {
00020     graph_t* g;
00021     g = (graph_t*) malloc(sizeof(graph_t));
00022     assert(g);
00023     if (g != NULL) {
```

```
00024         g->type = type;  /* initialize type of graph */
00025         g->n = n;        /* initialize number of vertices */
00026         g->m = 0;        /* g is a forest => 0 edge */
00027         g->vertex = (gvertex_t**) malloc(sizeof(gvertex_t*) * n);
00028         assert(g->vertex);
00029         for(int i = 0; i < g->n ; i++) {
00030             g->vertex[i] = (gvertex_t*) malloc(sizeof(gvertex_t));
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);
00035             g->vertex[i]->adj_list_len = 5;
00036             /* g is a forest => deg_in(i) = deg_out(i) = 0 */
00037             g->vertex[i]->degree.in = 0;
00038             g->vertex[i]->degree.out = 0;
00039         }
00040     }
00041     return g;
00042 }
00043
00048 void graph_destruct(graph_t* g) {
00049     for (int i = 0; i < g->n ; i++) {
00050         free(g->vertex[i]->adj_list);
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) {
00079     gvertex_t *v_u = g->vertex[u], *v_v = g->vertex[v];
00080     assert(u >= 0 && v >= 0 && u < g->n && v < g->n);
00081     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) {
00084         int* newlist = realloc(v_u->adj_list, sizeof(int) * (v_u->
      adj_list_len + 10));
00085         assert(newlist);
00086         v_u->adj_list = newlist;
00087         v_u->adj_list_len += 10;
00088     }
00089     v_u->adj_list[v_u->degree.out] = v;
00090     v_u->degree.out++;
00091     v_v->degree.in++;
00092     if (g->type == UNDIGRAPH && u != v) {
00093         assert(v_v->degree.out <= v_v->adj_list_len);
00094         if (v_v->degree.out == v_v->adj_list_len) {
00095             int* newlist = realloc(v_v->adj_list, sizeof(int) * (v_v->adj_list_len + 10));
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 }
00106
00111 void graph_adj_list_print(graph_t* g) {
00112     if (g->type == DIGRAPH) printf("type : digraph\n");
00113     else if (g->type == UNDIGRAPH) printf("type : undigraph\n");
00114     printf("n=%d, m=%d\n", g->n, g->m);
00115     for(int i = 0; i < g->n; i++) {
00116         printf("[%d] ", i);
00117         if (g->vertex[i]->degree.out > 0) printf("-> ");
00118         for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
00119             printf("[%d]", g->vertex[i]->adj_list[j]);
00120         }
00121         printf("\n");
00122     }
00123 }
00124
00131 graph_t* graph_clone(graph_t* g) {
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;
```

```
00136         new_g->n = g->n;
00137         new_g->m = g->m;
00138         new_g->vertex = (gvertex_t**) malloc(sizeof(g->vertex));
00139         assert(new_g->vertex);
00140         for(int i = 0; i < g->n ; i++) {
00141             new_g->vertex[i] = (gvertex_t*) malloc(sizeof(
      gvertex_t));
00142             assert(g->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);
00145             new_g->vertex[i]->adj_list_len = g->vertex[i]->
      adj_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];
00161     assert(v_u && v_v);
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;
00166             for (int j = i; j < v_u->degree.out - 1; j++) {
00167                 v_u->adj_list[j] = v_u->adj_list[j + 1];
00168             }
00169             v_u->degree.out--;
00170         }
00171     }
00172     if (g->type == UNDIGRAPH && flag) {
00173         for (int i = 0; i < v_v->degree.out; i++) {
00174             if(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* g, char* filename) {
00193     FILE* pfile;
00194     pfile = fopen(filename, "w");
00195     if (pfile == NULL) perror ("Error opening file");
00196     if (g->type == DIGRAPH) {
00197         fprintf(pfile, "digraph g {\nnode [shape=\"circle\"];\n");
00198         for(int i = 0; i < g->n; i++) {
00199             if(g->vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
00200             for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
00201                 fprintf(pfile, "%d -> %d;\n", i, g->vertex[i]->adj_list[j]);
00202             }
00203         }
00204     } else if (g->type == UNDIGRAPH) {
00205         fprintf(pfile, "graph g {\nnode [shape=\"circle\"];\n");
00206         for(int i = 0; i < g->n; i++) {
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++) {
00209                 if(i <= g->vertex[i]->adj_list[j]) {
00210                     fprintf(pfile, "%d -- %d;\n", i, g->vertex[i]->
      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) {
00221     int* color = malloc(g->n * sizeof(int));
00222     int* pred  = malloc(g->n * sizeof(int));
00223     int* dist  = malloc(g->n * sizeof(int));
00224     assert(color && pred && dist);
00225     for(int i = 0; i < g->n; i++) {
00226         color[i] = 0;
00227         pred[i] = -1;
00228         dist[i] = g->n + 1;
00229     }
```

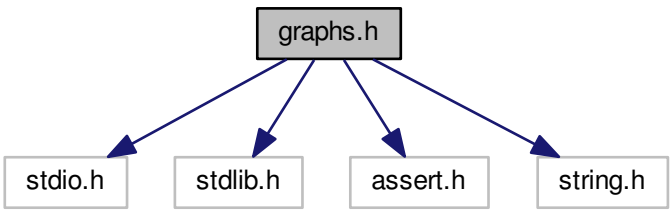
```
00230     queue_t* q = queue_create(sizeof(int), g->n);
00231     assert(q);
00232     int i = src;
00233     for (int j = 0; j < g->n; j++) {
00234         if (color[i] == 0) {
00235             color[i] = 1;
00236             dist[0];
00237             pred[i] = -1;
00238             queue_enqueue(q,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

6.3 graphs.h File Reference

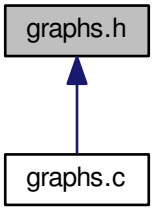
graph definition and basic operations

```
#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 graph

DIGRAPH direct graph

Definition at line 18 of file graphs.h.

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

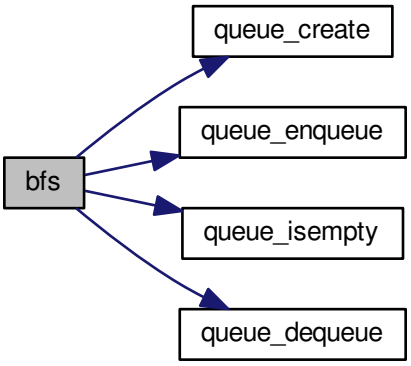
6.3.3 Function Documentation

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

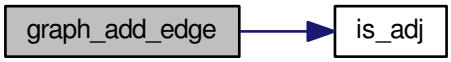
Parameters

g	graph
u	vertex index
v	vertex index

Definition at line 78 of file graphs.c.

```
00078                                           {
00079     gvertex_t *v_u = g->vertex[u], *v_v = g->vertex[v];
00080     assert(u >= 0 && v >= 0 && u < g->n && v < g->n);
00081     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) {
00084         int* newlist = realloc(v_u->adj_list, sizeof(int) * (v_u->
      adj_list_len + 10));
00085         assert(newlist);
00086         v_u->adj_list = newlist;
00087         v_u->adj_list_len += 10;
00088     }
00089     v_u->adj_list[v_u->degree.out] = v;
00090     v_u->degree.out++;
00091     v_v->degree.in++;
00092     if (g->type == UNDIGRAPH && u != v) {
00093         assert(v_v->degree.out <= v_v->adj_list_len);
00094         if (v_v->degree.out == v_v->adj_list_len) {
00095             int* newlist = realloc(v_v->adj_list, sizeof(int) * (v_v->adj_list_len + 10));
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.

```
00111                                       {
00112     if (g->type == DIGRAPH) printf("type : digraph\n");
00113     else if (g->type == UNDIGRAPH) printf("type : undigraph\n");
00114     printf("n=%d, m=%d\n", g->n, g->m);
00115     for(int i = 0; i < g->n; i++) {
00116         printf("[%d] ", i);
00117         if (g->vertex[i]->degree.out > 0) printf("-> ");
00118         for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
00119             printf("[%d]", g->vertex[i]->adj_list[j]);
00120         }
00121         printf("\n");
00122     }
00123 }
```

6.3.3.4 graph_t* graph_clone (graph_t * g)

Clone a graph

Parameters

g	graph

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;
00136         new_g->n = g->n;
00137         new_g->m = g->m;
00138         new_g->vertex = (gvertex_t**) malloc(sizeof(g->vertex));
00139         assert(new_g->vertex);
00140         for(int i = 0; i < g->n ; i++) {
00141             new_g->vertex[i] = (gvertex_t*) malloc(sizeof(
      gvertex_t));
00142             assert(g->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);
00145             new_g->vertex[i]->adj_list_len = g->vertex[i]->
      adj_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 }
```

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
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) {
00024         g->type = type;  /* initialize type of graph */
00025         g->n = n;        /* initialize number of vertices */
00026         g->m = 0;        /* g is a forest => 0 edge */
00027         g->vertex = (gvertex_t**) malloc(sizeof(gvertex_t*) * n);
00028         assert(g->vertex);
00029         for(int i = 0; i < g->n ; i++) {
00030             g->vertex[i] = (gvertex_t*) malloc(sizeof(gvertex_t));
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);
00035             g->vertex[i]->adj_list_len = 5;
00036             /* g is a forest => deg_in(i) = deg_out(i) = 0 */
00037             g->vertex[i]->degree.in = 0;
00038             g->vertex[i]->degree.out = 0;
00039         }
00040     }
00041     return g;
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
u	vertex index
v	vertex index

Definition at line 159 of file graphs.c.

```
00159                                                       {
00160     gvertex_t *v_u = g->vertex[u], *v_v = g->vertex[v];
00161     assert(v_u && v_v);
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;
00166             for (int j = i; j < v_u->degree.out - 1; j++) {
00167                 v_u->adj_list[j] = v_u->adj_list[j + 1];
00168             }
00169             v_u->degree.out--;
00170         }
00171     }
00172     if (g->type == UNDIGRAPH && flag) {
00173         for (int i = 0; i < v_v->degree.out; i++) {
00174             if(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 }
```

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.

```
00048                                   {
00049     for (int i = 0; i < g->n ; i++) {
00050         free(g->vertex[i]->adj_list);
00051         free(g->vertex[i]);
00052     }
00053     free(g->vertex);
00054     free(g);
00055 }
```

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

Todo system call ?

Definition at line 192 of file graphs.c.

```
00192                                                {
00193     FILE* pfile;
00194     pfile = fopen(filename, "w");
00195     if (pfile == NULL) perror ("Error opening file");
00196     if (g->type == DIGRAPH) {
00197         fprintf(pfile, "digraph g {\nnode [shape=\"circle\"];\n");
00198         for(int i = 0; i < g->n; i++) {
00199             if(g->vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
00200             for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
00201                 fprintf(pfile, "%d -> %d;\n", i, g->vertex[i]->adj_list[j]);
00202             }
00203         }
00204     } else if (g->type == UNDIGRAPH) {
00205         fprintf(pfile, "graph g {\nnode [shape=\"circle\"];\n");
00206         for(int i = 0; i < g->n; i++) {
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++) {
00209                 if(i <= g->vertex[i]->adj_list[j]) {
00210                     fprintf(pfile, "%d -- %d;\n", i, g->vertex[i]->
      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
u	vertex index
v	vertex index

Definition at line 63 of file graphs.c.

```
00063                                       {
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 }
```

6.4 graphs.h

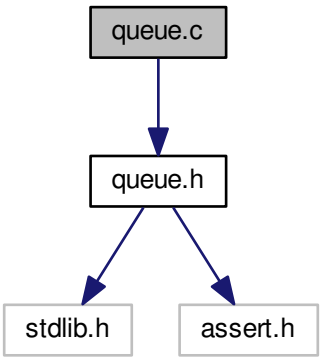
```
00001 #ifndef GRAPHS_H
00002 #define GRAPHS_H
00003
00012 #include <stdio.h>
00013 #include <stdlib.h>
00014 #include <assert.h>
00015 #include <string.h>
00016
00018 typedef enum {
00019     UNDIGRAPH,
```

```
00020     DIGRAPH
00021 } gtype_t;
00022
00023 typedef struct deg_t {
00024     int in;
00025     int out;
00026 } deg_t;
00027
00028 typedef struct gvertex_t {
00029     int* adj_list;
00030     int  adj_list_len;
00031     deg_t degree;
00032 } gvertex_t;
00033
00034 typedef struct graph_t {
00035     gtype_t type;
00036     int n;
00037     int m;
00038     gvertex_t** vertex;
00039 } graph_t;
00040
00041 typedef struct info_t {
00042     int* pred;
00043     int* dist;
00044 } info_t;
00045
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_enqueue for more information on the behavior in the excess case.

Definition at line 60 of file queue.c.

```
00060                                           {
00061     queue_t* q = malloc(sizeof(queue_t));
00062     assert(q);
```

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

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.

```
00043                                {
00044     void* e = q->base[(q->front - q->count + q->max_size)%q->
      max_size];
00045     q->count--;
00046     return e;
00047 }
```

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
e	element which be enqueued

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.

```
00030                                       {
00031     q->base[q->front] = e;
00032     if (q->front == q->max_size - 1) q->front = 0;
00033     else q->front++;
00034     q->count = (q->count + 1) % q->max_size;
00035 }
```

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.

```
00017                                {
00018     return q->count == 0;
00019 }
```

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) {
00031     q->base[q->front] = e;
00032     if (q->front == q->max_size - 1) q->front = 0;
00033     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];
00045     q->count--;
00046     return e;
00047 }
00048
00060 queue_t* queue_create(size_t width, int max_size) {
00061     queue_t* q = malloc(sizeof(queue_t));
00062     assert(q);
00063     q->width = width;
00064     q->max_size = max_size ;
00065     q->base = (void**) calloc(q->max_size, sizeof(void*));
00066     assert(q->base);
00067     q->front = 0;
00068     q->count = 0;
```

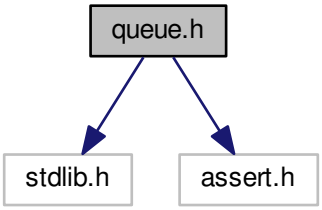
```
00069     return q;
00070 }
00071
00077 void queue_destruct(queue_t* q) {
00078     free(q->base);
00079     free(q);
00080 }
00081
00082
00083
```

6.7 queue.h File Reference

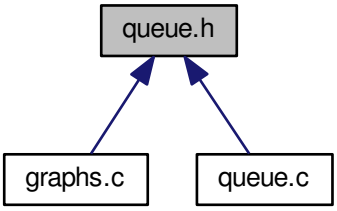
queue (using array) definition and basic operations

```
#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.

```
00060                                                   {
00061     queue_t* q = malloc(sizeof(queue_t));
00062     assert(q);
00063     q->width = width;
00064     q->max_size = max_size ;
00065     q->base = (void**) calloc(q->max_size, sizeof(void*));
00066     assert(q->base);
00067     q->front = 0;
00068     q->count = 0;
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.

```
00043                                   {
00044     void* e = q->base[(q->front - q->count + q->max_size)%q->
      max_size];
00045     q->count--;
00046     return e;
00047 }
```

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);
00080 }
```

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

Enqueue an element e into the queue q.

Parameters

q	queue
e	element which be enqueued

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.

```
00030                                       {
00031     q->base[q->front] = e;
00032     if (q->front == q->max_size - 1) q->front = 0;
00033     else q->front++;
00034     q->count = (q->count + 1) % q->max_size;
00035 }
```

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.

```
00017                                {
00018     return q->count == 0;
00019 }
```

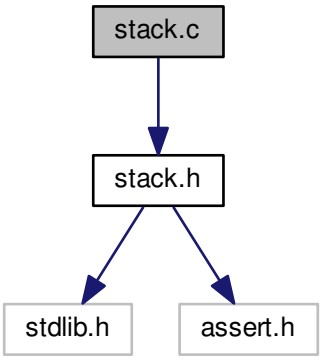
6.8 queue.h

```
00001 #ifndef STACK_H
00002 #define STACK_H
00003
00004 #include <stdlib.h>
00005 #include <assert.h>
00006
00021 typedef struct {
00022     size_t width;
00023     int front;
00024     int count;
00025     void** base;
00026     int max_size;
00027 } queue_t;
00028
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 element

Returns

a stack initialized

Definition at line 57 of file stack.c.

```
00057                                       {
00058     stack_t* s = malloc(sizeof(stack_t));
00059     assert(s);
00060     s->width = width;
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 }
```

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.

```
00073                                     {
00074     free(s->base);
00075     free(s);
00076 }
```

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.

```
00017                                   {
00018     return s->top == -1;
00019 }
```

6.9.2.4 void* stack_pop (stack_t * s)

Pop out an element from the stack s.

Parameters

s	stack

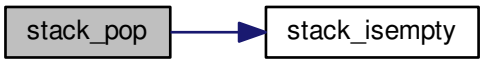
Returns

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
e	element which be pushed

Definition at line 27 of file stack.c.

```
00027                                   {
00028     s->top++;
00029     if (s->top == s->mem_size) {
00030         void** newptr = realloc(s->base, sizeof(void*) * (s->mem_size + 10));
00031         assert(newptr);
00032         s->base = newptr;
00033         s->mem_size += 10;
00034     }
00035     s->base[s->top] = e;
00036 }
```

6.10 stack.c

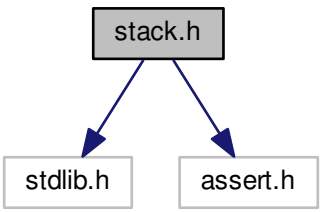
```
00001
00009 #include "stack.h"
00010
00017 int stack_isempty(stack_t* s) {
00018     return s->top == -1;
00019 }
00020
00027 void stack_push(stack_t* s, void* e) {
00028     s->top++;
00029     if (s->top == s->mem_size) {
00030         void** newptr = realloc(s->base, sizeof(void*) * (s->mem_size + 10));
00031         assert(newptr);
00032         s->base = newptr;
00033         s->mem_size += 10;
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) {
00058     stack_t* s = malloc(sizeof(stack_t));
00059     assert(s);
00060     s->width = width;
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

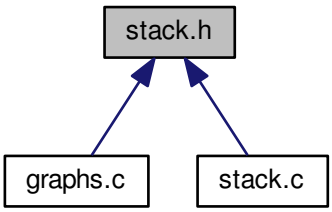
stack definition and basic operations

```
#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 element

Returns

a stack initialized

Definition at line 57 of file stack.c.

```
00057                                       {
00058     stack_t* s = malloc(sizeof(stack_t));
00059     assert(s);
00060     s->width = width;
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 }
```

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.

```
00073                                     {
00074     free(s->base);
00075     free(s);
00076 }
```

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.

```
00017                                    {
00018     return s->top == -1;
00019 }
```

6.11.2.4 void* stack_pop (stack_t * s)

Pop out an element from the stack s.

Parameters

s	stack

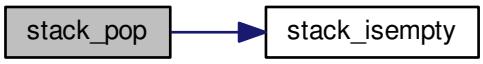
Returns

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
e	element which be pushed

Definition at line 27 of file stack.c.

```
00027                                   {
00028     s->top++;
00029     if (s->top == s->mem_size) {
00030         void** newptr = realloc(s->base, sizeof(void*) * (s->mem_size + 10));
00031         assert(newptr);
00032         s->base = newptr;
00033         s->mem_size += 10;
00034     }
00035     s->base[s->top] = e;
00036 }
```

6.12 stack.h

```
00001 #ifndef STACK_H
00002 #define STACK_H
00003
00004 #include <stdlib.h>
00005 #include <assert.h>
00006
00020 typedef struct {
00021     size_t width;
00022     int top;
00023     void** base;
00024     int mem_size;
00025 } stack_t;
00026
00027 stack_t* stack_create(size_t width);
00028 void stack_destruct(stack_t* s);
00029 int stack_isempty(stack_t* s);
00030 void* stack_pop(stack_t* s);
00031 void stack_push(stack_t* s, void* e);
00032
00033 #endif /* ifndef STACK_H */
```

Index