Graph

Generated by Doxygen 1.8.11

Sun Dec 31 2017 22:19:06

Contents

1 Todo List

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.

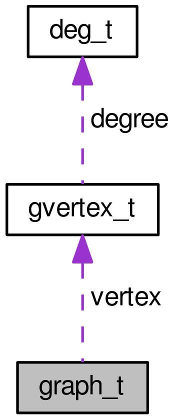
The documentation for this struct was generated from the following file:

- 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.

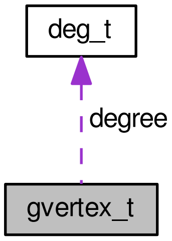
The documentation for this struct was generated from the following file:

- 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 src
- 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 44 of file graphs.h.

5.4.2.2 int∗ pred

Definition at line 43 of file graphs.h.

5.4.2.3 int src

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 22 of file queue.h.

5.5.2 Field Documentation

5.5.2.1 void** base

pointer to the array

Definition at line 26 of file queue.h.

5.5.2.2 int count

count element amount

Definition at line 25 of file queue.h.

5.5.2.3 int front

front element index

Definition at line 24 of file queue.h.

5.5.2.4 int max_size

width ∗ max_size bytes is reserved for the queue

Definition at line 27 of file queue.h.

5.5.2.5 size_t width

element size (in bytes)

Definition at line 23 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

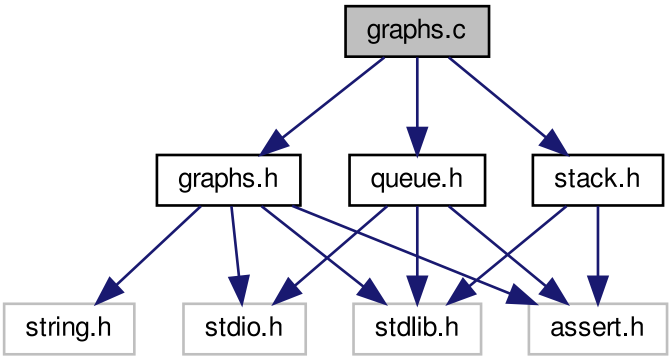
6 File Documentation

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)
- void graph_search_dot_output (graph_t ∗g, info_t ∗info, char ∗filename)
- info_t ∗ bfs (graph_t ∗g, int src)
- void info_destruct (info_t ∗info)

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*)

Launch a bfs on the given graph and source vertex.

Parameters

g	graph
src	source vertex

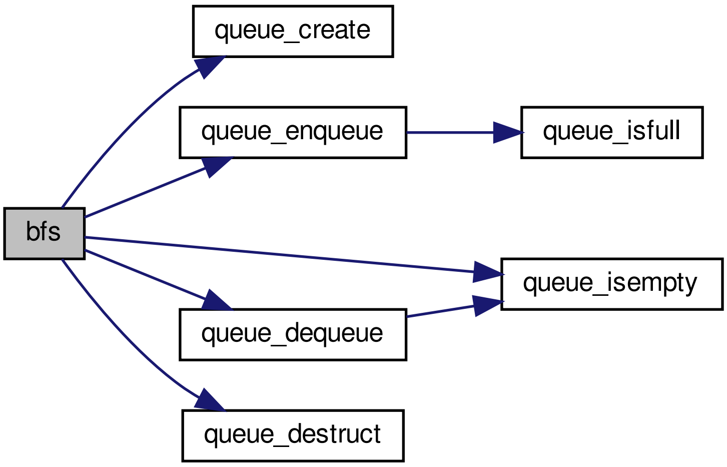
Returns

search info which contains for each vertex the distance to the source and the index of its predecessor if it exists (if not, info->pred[i] = -1)

Definition at line 278 of file graphs.c.

```
                                 {
    assert(src >= 0 && src < g->n);
    int* color = malloc(g->n * sizeof(int));
    int* pred  = malloc(g->n * sizeof(int));
    int* dist  = malloc(g->n * sizeof(int));
    assert(color && pred && dist);
    for(int i = 0; i < g->n; i++) {
        color[i] = 0;       /* white                 */
        pred[i] = -1;       /* no predecessor        */
        dist[i] = g->n + 1; /* no reachable distance */
    }
    color[src] = 1;
    dist[src] = 0;          /* dist(src,src) = 0     */
    /* int-queue with n slots */
    queue_t* q = queue_create(sizeof(int), g->n);
    queue_enqueue(q, inttoptr(src));
    while(!queue_isempty(q)) {
        int* u = queue_dequeue(q);
        for(int i = 0; i < g->vertex[*u]->degree.out; i++) {
            int v = g->vertex[*u]->adj_list[i];
            if (color[v] == 0) {
                color[v] = 1; /* gray */
                dist[v] = dist[*u] + 1;
                pred[v] = *u;
                queue_enqueue(q, inttoptr(v));
            }
        }
        color[*u] = 2; /* black */
        free(u);
    }
    queue_destruct(q);
    free(color);
    info_t* info = malloc(sizeof(info_t));
    assert(info);
    info->src = src;
    info->dist = dist;
    info->pred = pred;
    return info;
}
```

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.

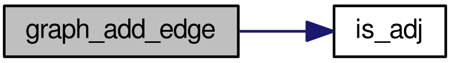
Parameters

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).

Parameters

g	graph
filename	filename without any extension

Definition at line 191 of file graphs.c.

```
00191                                                   {
00192     FILE* pfile;
00193     pfile = fopen(filename, "w");
00194     if (pfile == NULL) perror ("Error opening file");
00195     if (g->type == DIGRAPH) {
00196         fprintf(pfile, "digraph g {\nnode [shape=\"circle\"];\ngraph [overlap=false, concentrate=true];\n")
      ;
00197         for(int i = 0; i < g->n; i++) {
00198             if(g->vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
00199             for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
00200                 fprintf(pfile, "%d -> %d;\n", i, g->vertex[i]->adj_list[j]);
00201             }
00202         }
00203     } else if (g->type == UNDIGRAPH) {
00204         fprintf(pfile, "graph g {\nnode [shape=\"circle\"];\ngraph [overlap=false, concentrate=true];\n");
00205         for(int i = 0; i < g->n; i++) {
00206             if(g->vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
00207             for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
00208                 if(i <= g->vertex[i]->adj_list[j]) {
00209                     fprintf(pfile, "%d -- %d;\n", i, g->vertex[i]->
      adj_list[j]);
00210                 }
00211             }
00212         }
00213     }
00214     fprintf(pfile, "}\n");
00215     fclose(pfile);
00216 }
```

6.1.2.9 void graph_search_dot_output (graph_t * g, info_t * info, char * filename)

Output the graph g with bfs/dfs edge and reachable colored in dot format (filename.dot).

Parameters

g	graph
info	search info, result of a bfs/dfs
filename	filename without any extension

Definition at line 225 of file graphs.c.

```
00225                                                                       {
00226     FILE* pfile;
00227     pfile = fopen(filename, "w");
00228     if (pfile == NULL) perror ("Error opening file");
00229     if (g->type == DIGRAPH) {
00230         fprintf(pfile, "digraph g {\nnode [shape=\"circle\"];\ngraph [overlap=false, concentrate=true];\n");
00231         fprintf(pfile, "%d [style=filled fillcolor=red, peripheries=2];\n", info->src);
00232         for(int i = 0; i < g->n; i++) {
00233             if(g->vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
00234             for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
00235                 if(info->pred[g->vertex[i]->adj_list[j]] == i) {
00236                     fprintf(pfile, "%d -> %d[color=blue];\n", i, g->vertex[i]->adj_list[j]);
00237                     fprintf(pfile, "%d [style=filled fillcolor=red];\n", g->vertex[i]->adj_list[j]);
00238                 } else fprintf(pfile, "%d -> %d;\n", i, g->vertex[i]->adj_list[j]);
00239             }
00240         }
00241     } else if (g->type == UNDIGRAPH) {
00242         fprintf(pfile, "graph g {\nnode [shape=\"circle\"];\ngraph [overlap=false,concentrate=true];\n");
00243         fprintf(pfile, "%d [style=filled fillcolor=red, peripheries=2];\n", info->src);
00244         for (int i = 0; i < g->n ; i++) {
00245             if(info->pred[i] == -1) continue;
00246             fprintf(pfile, "%d -- %d[color=blue];\n", info->pred[i], i);
00247             fprintf(pfile, "%d [style=filled fillcolor=red];\n", i);
00248         }
00249         for (int i = 0; i < g->n ; i++) {
00250             if(g->vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
00251             for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
00252                 if(i <= g->vertex[i]->adj_list[j]) {
00253                     if (info->pred[i] == g->vertex[i]->adj_list[j]) continue;
00254                     else fprintf(pfile, "%d -- %d;\n", i, g->vertex[i]->adj_list[j]);
00255                 }
00256             }
00257         }
00258     }
00259     fprintf(pfile, "}\n");
00260     fclose(pfile);
00261 }
```

6.1.2.10 void info_destruct (info_t ∗ *info*)

Free search info

Parameters

info	search info

Definition at line 324 of file graphs.c.

```
00324                                 {
00325     free(info->dist);
00326     free(info->pred);
00327     free(info);
00328 }
```

6.1.2.11 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];
```

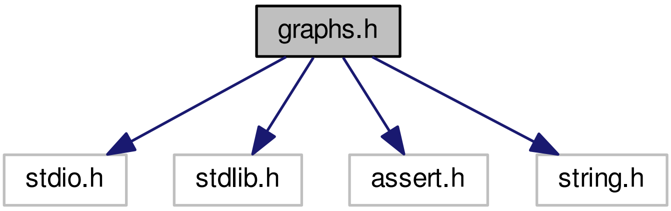
```
    assert(u >= 0 && v >= 0 && u < g->n && v < g->n);
    assert(v_u->degree.out <= v_u->adj_list_len);
    if (is_adj(g, u, v)) return ;
    if (v_u->degree.out == v_u->adj_list_len) {
        int* newlist = realloc(v_u->adj_list, sizeof(int) * (v_u->adj_list_len + 10));
        assert(newlist);
        v_u->adj_list = newlist;
        v_u->adj_list_len += 10;
    }
    v_u->adj_list[v_u->degree.out] = v;
    v_u->degree.out++;
    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) {
            int* newlist = realloc(v_v->adj_list, sizeof(int) * (v_v->adj_list_len + 10));
            assert(newlist);
            v_v->adj_list = newlist;
            v_v->adj_list_len += 10;
        }
        v_v->adj_list[v_v->degree.out] = u;
        v_v->degree.out++;
        v_u->degree.in++;
    }
    g->m++;
}

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");
    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("-> ");
        for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
            printf("[%d]", g->vertex[i]->adj_list[j]);
        }
        printf("\n");
    }
}

graph_t* graph_clone(graph_t* g) {
    graph_t* new_g;
    new_g = (graph_t*) malloc(sizeof(graph_t));
    if (new_g != NULL) {
        new_g->type = g->type;
        new_g->n = g->n;
        new_g->m = g->m;
        new_g->vertex = (gvertex_t**) malloc(sizeof(g->vertex));
        assert(new_g->vertex);
        for(int i = 0; i < g->n ; i++) {
            new_g->vertex[i] = (gvertex_t*) malloc(sizeof(gvertex_t));
            assert(g->vertex[i]);
            new_g->vertex[i]->adj_list = (int*) malloc(sizeof(g->vertex[i]->adj_list));
            assert(new_g->vertex[i]->adj_list);
            new_g->vertex[i]->adj_list_len = g->vertex[i]->adj_list_len;
            new_g->vertex[i]->degree.in = g->vertex[i]->degree.in;
            new_g->vertex[i]->degree.out = g->vertex[i]->degree.out;
        }
    }
    return new_g;
}

void graph_delete_adj_ele(graph_t* g, int u, int v) {
    gvertex_t *v_u = g->vertex[u], *v_v = g->vertex[v];
    assert(v_u && v_v);
    int flag = 0;
    for (int i = 0; i < v_u->degree.out; i++) {
        if(v_u->adj_list[i] == v) {
            flag = 1;
            for (int j = i; j < v_u->degree.out - 1; j++) {
                v_u->adj_list[j] = v_u->adj_list[j + 1];
            }
            v_u->degree.out--;
        }
    }
    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];
                }
                v_v->degree.out--;
            }
        }
    }
    if(flag) g->m--;
}

void graph_dot_output(graph_t* g, char* filename) {
    FILE* pfile;
    pfile = fopen(filename, "w");
    if (pfile == NULL) perror ("Error opening file");
    if (g->type == DIGRAPH) {
        fprintf(pfile, "digraph g {\nnode [shape=\"circle\"];\ngraph [overlap=false, concentrate=true];\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]);
            }
        }
    } else if (g->type == UNDIGRAPH) {
        fprintf(pfile, "graph g {\nnode [shape=\"circle\"];\ngraph [overlap=false, concentrate=true];\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++) {
                if(i <= g->vertex[i]->adj_list[j]) {
                    fprintf(pfile, "%d -- %d;\n", i, g->vertex[i]->adj_list[j]);
                }
            }
        }
    }
    fprintf(pfile, "}\n");
    fclose(pfile);
}

void graph_search_dot_output(graph_t* g, info_t* info, char* filename)
{
    FILE* pfile;
    pfile = fopen(filename, "w");
    if (pfile == NULL) perror ("Error opening file");
    if (g->type == DIGRAPH) {
        fprintf(pfile, "digraph g {\nnode [shape=\"circle\"];\ngraph [overlap=false, concentrate=true];\n");
        fprintf(pfile, "%d [style=filled fillcolor=red, peripheries=2];\n", info->src);
        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++) {
                if(info->pred[g->vertex[i]->adj_list[j]] == i) {
                    fprintf(pfile, "%d -> %d[color=blue];\n", i, g->vertex[i]->adj_list[j]);
                    fprintf(pfile, "%d [style=filled fillcolor=red];\n", g->vertex[i]->adj_list[j]);
                } else fprintf(pfile, "%d -> %d;\n", i, g->vertex[i]->adj_list[j]);
            }
        }
    } else if (g->type == UNDIGRAPH) {
        fprintf(pfile, "graph g {\nnode [shape=\"circle\"];\ngraph [overlap=false,concentrate=true];\n");
        fprintf(pfile, "%d [style=filled fillcolor=red, peripheries=2];\n", info->src);
        for (int i = 0; i < g->n ; i++) {
            if(info->pred[i] == -1) continue;
            fprintf(pfile, "%d -- %d[color=blue];\n", info->pred[i], i);
            fprintf(pfile, "%d [style=filled fillcolor=red];\n", i);
        }
        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++) {
                if(i <= g->vertex[i]->adj_list[j]) {
                    if (info->pred[i] == g->vertex[i]->adj_list[j]) continue;
                    else fprintf(pfile, "%d -- %d;\n", i, g->vertex[i]->adj_list[j]);
                }
            }
        }
    }
    fprintf(pfile, "}\n");
    fclose(pfile);
}

static int* inttoptr(int i) {
    int* ptr = malloc(sizeof(int));
    assert(ptr);
    *ptr = i;
    return ptr;
}

info_t* bfs(graph_t* g, int src) {
    assert(src >= 0 && src < g->n);
    int* color = malloc(g->n * sizeof(int));
    int* pred  = malloc(g->n * sizeof(int));
    int* dist  = malloc(g->n * sizeof(int));
    assert(color && pred && dist);
    for(int i = 0; i < g->n; i++) {
        color[i] = 0;       /* white                 */
        pred[i] = -1;       /* no predecessor        */
        dist[i] = g->n + 1; /* no reachable distance */
    }
    color[src] = 1;
    dist[src] = 0;          /* dist(src,src) = 0     */
    /* int-queue with n slots */
    queue_t* q = queue_create(sizeof(int), g->n);
    queue_enqueue(q, inttoptr(src));
    while(!queue_isempty(q)) {
        int* u = queue_dequeue(q);
        for(int i = 0; i < g->vertex[*u]->degree.out; i++) {
            int v = g->vertex[*u]->adj_list[i];
            if (color[v] == 0) {
                color[v] = 1; /* gray */
                dist[v] = dist[*u] + 1;
                pred[v] = *u;
                queue_enqueue(q, inttoptr(v));
            }
        }
        color[*u] = 2; /* black */
        free(u);
    }
    queue_destruct(q);
    free(color);
    info_t* info = malloc(sizeof(info_t));
    assert(info);
    info->src = src;
    info->dist = dist;
    info->pred = pred;
    return info;
}


void info_destruct(info_t* info) {
    free(info->dist);
    free(info->pred);
    free(info);
}
```

6.3 graphs.h File Reference

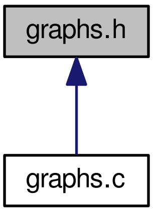
graph definition and basic operations

```
#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)
- void graph_search_dot_output (graph_t ∗g, info_t ∗info, char ∗filename)
- int is_adj (graph_t ∗g, int u, int v)
- info_t ∗ bfs (graph_t ∗g, int src)
- void info_destruct (info_t ∗info)

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*)

Launch a bfs on the given graph and source vertex.

Parameters

g	graph
src	source vertex

Returns

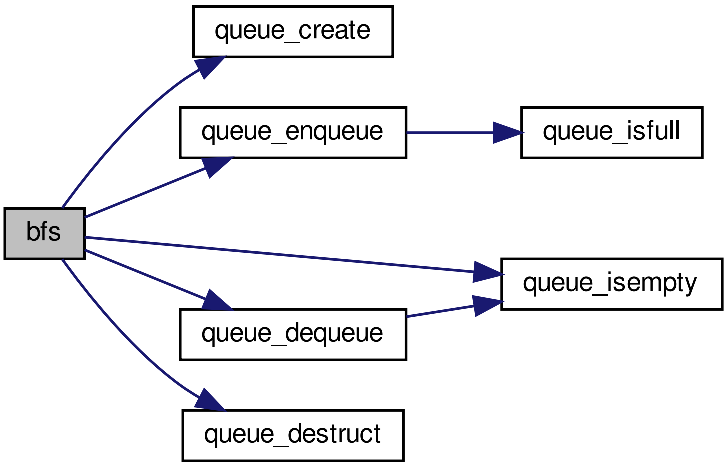
search info which contains for each vertex the distance to the source and the index of its predecessor if it exists (if not, info->pred[i] = -1)

Definition at line 278 of file graphs.c.

```
00278                                    {
00279     assert(src >= 0 && src < g->n);
00280     int* color = malloc(g->n * sizeof(int));
00281     int* pred  = malloc(g->n * sizeof(int));
00282     int* dist  = malloc(g->n * sizeof(int));
00283     assert(color && pred && dist);
00284     for(int i = 0; i < g->n; i++) {
00285         color[i] = 0;        /* white               */
00286         pred[i] = -1;        /* no predecessor      */
00287         dist[i] = g->n + 1;  /* no reachable distance */
00288     }
00289     color[src] = 1;
00290     dist[src] = 0;           /* dist(src,src) = 0     */
00291     /* int-queue with n slots */
00292     queue_t* q = queue_create(sizeof(int), g->n);
00293     queue_enqueue(q, inttoptr(src));
00294     while(!queue_isempty(q)) {
00295         int* u = queue_dequeue(q);
00296         for(int i = 0; i < g->vertex[*u]->degree.out; i++) {
00297             int v = g->vertex[*u]->adj_list[i];
00298             if (color[v] == 0) {
00299                 color[v] = 1; /* gray */
00300                 dist[v] = dist[*u] + 1;
00301                 pred[v] = *u;
00302                 queue_enqueue(q, inttoptr(v));
00303             }
00304         }
```

```
        color[*u] = 2; /* black */
        free(u);
    }
    queue_destruct(q);
    free(color);
    info_t* info = malloc(sizeof(info_t));
    assert(info);
    info->src = src;
    info->dist = dist;
    info->pred = pred;
    return info;
}
```

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.

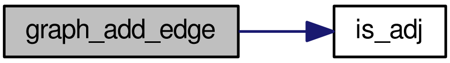
Parameters

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).

Parameters

g	graph
filename	filename without any extension

Definition at line 191 of file graphs.c.

```
00191                                                  {
00192     FILE* pfile;
00193     pfile = fopen(filename, "w");
00194     if (pfile == NULL) perror ("Error opening file");
00195     if (g->type == DIGRAPH) {
00196         fprintf(pfile, "digraph g {\nnode [shape=\"circle\"];\ngraph [overlap=false, concentrate=true];\n")
      ;
00197         for(int i = 0; i < g->n; i++) {
00198             if(g->vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
00199             for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
00200                 fprintf(pfile, "%d -> %d;\n", i, g->vertex[i]->adj_list[j]);
00201             }
00202         }
00203     } else if (g->type == UNDIGRAPH) {
00204         fprintf(pfile, "graph g {\nnode [shape=\"circle\"];\ngraph [overlap=false, concentrate=true];\n");
00205         for(int i = 0; i < g->n; i++) {
00206             if(g->vertex[i]->degree.out == 0) fprintf(pfile, "%d;\n", i);
00207             for(int j = 0; j < g->vertex[i]->degree.out ; j++) {
00208                 if(i <= g->vertex[i]->adj_list[j]) {
00209                     fprintf(pfile, "%d -- %d;\n", i, g->vertex[i]->
      adj_list[j]);
00210                 }
00211             }
00212         }
00213     }
00214     fprintf(pfile, "}\n");
00215     fclose(pfile);
00216 }
```

6.3.3.9 void graph_search_dot_output (graph_t * g, info_t * info, char * filename)

Output the graph g with bfs/dfs edge and reachable colored in dot format (filename.dot).

Parameters

g	graph
info	search info, result of a bfs/dfs
filename	filename without any extension

Definition at line 225 of file graphs.c.

```
00225                                                                   {
    FILE* pfile;
    pfile = fopen(filename, "w");
    if (pfile == NULL) perror ("Error opening file");
    if (g->type == DIGRAPH) {
        fprintf(pfile, "digraph g {\nnode [shape=\"circle\"];\ngraph [overlap=false, concentrate=true];\n");
        fprintf(pfile, "%d [style=filled fillcolor=red, peripheries=2];\n", info->src);
        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++) {
                if(info->pred[g->vertex[i]->adj_list[j]] == i) {
                    fprintf(pfile, "%d -> %d[color=blue];\n", i, g->vertex[i]->adj_list[j]);
                    fprintf(pfile, "%d [style=filled fillcolor=red];\n", g->vertex[i]->adj_list[j]);
                } else fprintf(pfile, "%d -> %d;\n", i, g->vertex[i]->adj_list[j]);
            }
        }
    } else if (g->type == UNDIGRAPH) {
        fprintf(pfile, "graph g {\nnode [shape=\"circle\"];\ngraph [overlap=false,concentrate=true];\n");
        fprintf(pfile, "%d [style=filled fillcolor=red, peripheries=2];\n", info->src);
        for (int i = 0; i < g->n ; i++) {
            if(info->pred[i] == -1) continue;
            fprintf(pfile, "%d -- %d[color=blue];\n", info->pred[i], i);
            fprintf(pfile, "%d [style=filled fillcolor=red];\n", i);
        }
        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++) {
                if(i <= g->vertex[i]->adj_list[j]) {
                    if (info->pred[i] == g->vertex[i]->adj_list[j]) continue;
                    else fprintf(pfile, "%d -- %d;\n", i, g->vertex[i]->adj_list[j]);
                }
            }
        }
    }
    fprintf(pfile, "}\n");
    fclose(pfile);
}
```

6.3.3.10 void info_destruct (info_t * *info*)

Free search info

Parameters

info	search info

Definition at line 324 of file graphs.c.

```
00324                                {
00325     free(info->dist);
00326     free(info->pred);
00327     free(info);
00328 }
```

6.3.3.11 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

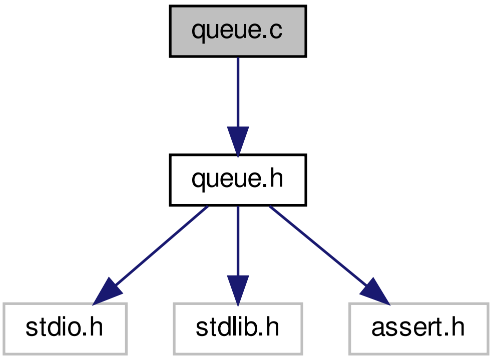
```
#ifndef GRAPHS_H
#define GRAPHS_H

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

typedef enum {
    UNDIGRAPH,
    DIGRAPH
} gtype_t;

typedef struct deg_t {
    int in;
    int out;
} deg_t;

typedef struct gvertex_t {
    int* adj_list;
    int  adj_list_len;
    deg_t degree;
} gvertex_t;

typedef struct graph_t {
    gtype_t type;
    int n;
    int m;
    gvertex_t** vertex;
} graph_t;

typedef struct info_t {
    int src;
    int* pred;
    int* dist;
} info_t;

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);
void graph_search_dot_output(graph_t* g, info_t* info, char* filename);
int is_adj(graph_t* g, int u, int v);
info_t* bfs(graph_t* g, int src);
void info_destruct(info_t* info);

#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)
- int queue_isfull (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

Definition at line 71 of file queue.c.

```
00071                                               {
00072     queue_t* q = malloc(sizeof(queue_t));
00073     assert(q);
00074     q->width = width;
00075     q->max_size = max_size ;
00076     q->base = (void**) calloc(q->max_size, sizeof(void*));
00077     assert(q->base);
00078     q->front = 0;
00079     q->count = 0;
00080     return q;
00081 }
```

6.5.2.2 void∗ queue_dequeue (queue_t ∗ *q*)

Dequeue an element from the queue s.

Parameters

q	queue

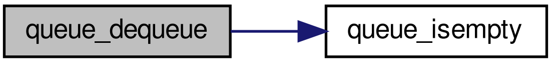
Returns

an element or NULL if the queue is empty

Definition at line 54 of file queue.c.

```
00054                            {
00055     if(queue_isempty(q)) {
00056         fprintf(stderr, "The queue is empty : failed to dequeue.\n");
00057         return NULL;
00058     }
00059     void* e = q->base[(q->front - q->count + q->max_size)%q->
      max_size];
00060     q->count--;
00061     return e;
00062 }
```

Here is the call graph for this function:



6.5.2.3 void queue_destruct (queue_t ∗ *q*)

Free a queue.

Parameters

q	a queue

Definition at line 88 of file queue.c.

```
00088                                 {
00089     free(q->base);
00090     free(q);
00091 }
```

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

Enqueue an element e into the queue q.

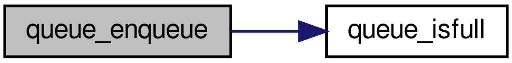
Parameters

q	queue
e	element which be enqueued

Definition at line 37 of file queue.c.

```
00037                                         {
00038     if(queue_isfull(q)) {
00039         fprintf(stderr, "The queue is full : failed to enqueue.\n");
00040         return;
00041     }
00042     q->base[q->front] = e;
00043     if (q->front == q->max_size - 1) q->front = 0;
00044     else q->front++;
00045     q->count = q->count + 1;
00046 }
```

Here is the call graph for this function:



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 }
```

6.5.2.6 int queue_isfull (queue_t ∗ *q*)

Determinate the fullness of a queue.

Parameters

s	queue

Returns

1 if the queue s is full, 0 otherwise.

Definition at line 27 of file queue.c.

```
00027                               {
00028     return q->count == q->max_size;
00029 }
```

6.6 queue.c

```
00001
00009 #include "queue.h"
00010
00017 int queue_isempty(queue_t* q) {
00018     return q->count == 0;
00019 }
00020
00027 int queue_isfull(queue_t* q) {
00028     return q->count == q->max_size;
00029 }
00030
00037 void queue_enqueue(queue_t* q, void* e) {
00038     if(queue_isfull(q)) {
00039         fprintf(stderr, "The queue is full : failed to enqueue.\n");
00040         return;
00041     }
00042     q->base[q->front] = e;
00043     if (q->front == q->max_size - 1) q->front = 0;
00044     else q->front++;
00045     q->count = q->count + 1;
00046 }
00047
00054 void* queue_dequeue(queue_t* q) {
00055     if(queue_isempty(q)) {
00056         fprintf(stderr, "The queue is empty : failed to dequeue.\n");
00057         return NULL;
```

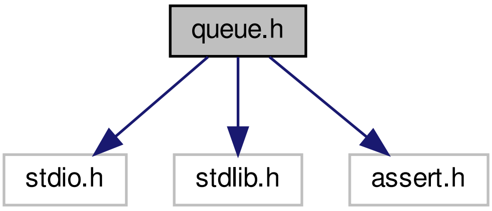
```
    }
    void* e = q->base[(q->front - q->count + q->max_size)%q->max_size];
    q->count--;
    return e;
}

queue_t* queue_create(size_t width, int max_size) {
    queue_t* q = malloc(sizeof(queue_t));
    assert(q);
    q->width = width;
    q->max_size = max_size ;
    q->base = (void**) calloc(q->max_size, sizeof(void*));
    assert(q->base);
    q->front = 0;
    q->count = 0;
    return q;
}

void queue_destruct(queue_t* q) {
    free(q->base);
    free(q);
}



```

6.7 queue.h File Reference

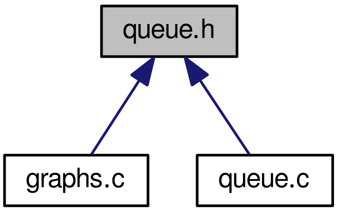
queue (using array) definition and basic operations

```
#include <stdio.h>
#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)
- int queue_isfull (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

Definition at line 71 of file queue.c.

```
00071                                              {
00072     queue_t* q = malloc(sizeof(queue_t));
00073     assert(q);
00074     q->width = width;
00075     q->max_size = max_size ;
00076     q->base = (void**) calloc(q->max_size, sizeof(void*));
00077     assert(q->base);
00078     q->front = 0;
00079     q->count = 0;
00080     return q;
00081 }
```

6.7.2.2 void∗ queue_dequeue (queue_t ∗ *q*)

Dequeue an element from the queue s.

Parameters

q	queue

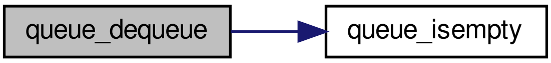
Returns

an element or NULL if the queue is empty

Definition at line 54 of file queue.c.

```
00054                                 {
00055     if(queue_isempty(q)) {
00056         fprintf(stderr, "The queue is empty : failed to dequeue.\n");
00057         return NULL;
00058     }
00059     void* e = q->base[(q->front - q->count + q->max_size)%q->
      max_size];
00060     q->count--;
00061     return e;
00062 }
```

Here is the call graph for this function:



6.7.2.3 void queue_destruct (queue_t ∗ *q*)

Free a queue.

Parameters

q	a queue

Definition at line 88 of file queue.c.

```
00088                                 {
00089     free(q->base);
00090     free(q);
00091 }
```

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

Enqueue an element e into the queue q.

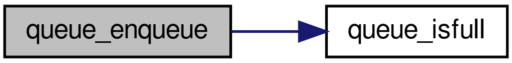
Parameters

q	queue
e	element which be enqueued

Definition at line 37 of file queue.c.

```
00037                                         {
00038     if(queue_isfull(q)) {
00039         fprintf(stderr, "The queue is full : failed to enqueue.\n");
00040         return;
00041     }
00042     q->base[q->front] = e;
00043     if (q->front == q->max_size - 1) q->front = 0;
00044     else q->front++;
00045     q->count = q->count + 1;
00046 }
```

Here is the call graph for this function:



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 }
```

6.7.2.6 int queue_isfull (queue_t ∗ *q*)

Determinate the fullness of a queue.

Parameters

s	queue

Returns

1 if the queue s is full, 0 otherwise.

Definition at line 27 of file queue.c.

```
00027                             {
00028     return q->count == q->max_size;
00029 }
```

6.8 queue.h

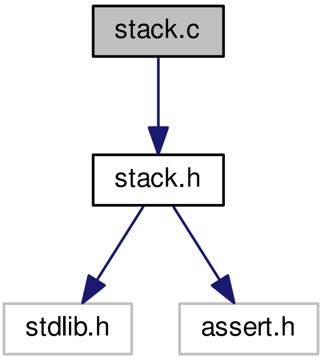
```
00001 #ifndef STACK_H
00002 #define STACK_H
00003 
00004 #include <stdio.h>
00005 #include <stdlib.h>
00006 #include <assert.h>
00007 
00022 typedef struct {
00023     size_t width;
00024     int front;
00025     int count;
00026     void** base;
00027     int max_size;
00028 } queue_t;
00029 
00030 queue_t* queue_create(size_t width, int max_size);
00031 void queue_destruct(queue_t* q);
00032 int queue_isempty(queue_t* q);
00033 int queue_isfull(queue_t* q);
00034 void* queue_dequeue(queue_t* q);
00035 void queue_enqueue(queue_t* q, void* e);
00036 
00037 #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

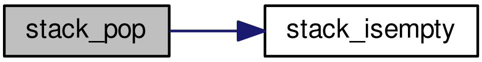
Returns

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

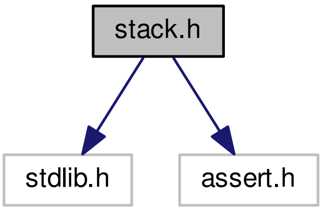
```

#include "stack.h"

int stack_isempty(stack_t* s) {
    return s->top == -1;
}

void stack_push(stack_t* s, void* e) {
    s->top++;
    if (s->top == s->mem_size) {
        void** newptr = realloc(s->base, sizeof(void*) * (s->mem_size + 10));
        assert(newptr);
        s->base = newptr;
        s->mem_size += 10;
    }
    s->base[s->top] = e;
}

void* stack_pop(stack_t* s) {
    if (stack_isempty(s)) return NULL;
    s->top--;
    return s->base[s->top + 1];
}

stack_t* stack_create(size_t width) {
    stack_t* s = malloc(sizeof(stack_t));
    assert(s);
    s->width = width;
    s->mem_size = 10;
    s->base = (void**) malloc(sizeof(void*) * s->mem_size);
    assert(s->base);
    s->top = -1;
    return s;
}

void stack_destruct(stack_t* s) {
    free(s->base);
    free(s);
}



```

6.11 stack.h File Reference

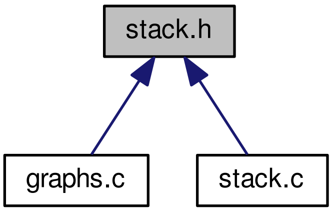
stack definition and basic operations

```
#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

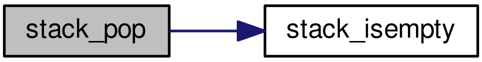
Returns

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

```c
#ifndef STACK_H
#define STACK_H

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

typedef struct {
    size_t width;
    int top;
    void** base;
    int mem_size;
} stack_t;

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);

#endif /* ifndef STACK_H */
```

Index