Assignment 4 - Software I, Spring 2004 (0368-2157-09/10/11/12)

`http://www.cs.tau.ac.il/~efif/courses/Software1_Spring_04`

Due: June 01, 2003

Required Knowledge arrays, structs, malloc, free, sort, and file IO.
Before starting to answer the questions, please read very carefully the "Submission Guidelines"[1]. Important: make sure your code uses meaningful names for variables and functions, and it is documented properly. Divide complex operations into simpler tasks, and implement each task in a separate function.

Ex 4 bfs

Write a program that performs a breadth-first search (BFS) in an undirected graph. The program reads a graph description from an input file, and a designated node r in this graph and constructs a tree (i.e., connected and acyclic subgraph) rooted at r, which contains all nodes reachable from r, using a BFS algorithm. Then, the program prints out the tree, clears the data structures, and exits. You need to implement the BFS algorithm, the data structures used, and a simple memory-management scheme.

Please download from `http://www.cs.tau.ac.il/~efif/courses/Software1_Spring_04/code/bfs` the following files:

- `bfs.h` - header file containing declarations of data structures and functions, as explained below.
- `bfs.c` - partial implementation — the `main` function and some utility functions, to get you started.
- `memory_man.h` - header file for memory management, included in `bfs.c`.
- `memory_man.c` - source code of memory management

You are required to use the data-structures defined in `bfs.h` and the functions declared in `bfs.h` and `memory_man.h`. You can extend these data structures and add new ones.

Command Line The syntax of the command line is:

```
bfs root filename
```

filename
: is the name of a graph-description file. The format of this file is described below.

root
: is the id of the node to be used as the start node of the BFS. The default is 0.

Graph Description The input file contains the graph description in the following format. The first line contains the number of nodes in the graph, denoted by *size* (use `unsigned int`). Each node has an id — a number between 0 and $size - 1$. The remaining lines describe the arcs, where each line defines a single arc in the form (a, b). Note that an arc is bi-directional, i.e., node a is a neighbor of b and vice versa. For example:

[1] `http://www.cs.tau.ac.il/~efif/courses/Software1_Spring_04/subgd.php`

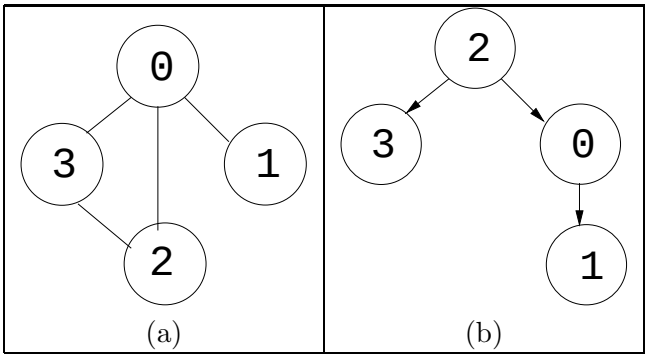

Figure 1: (a) an example of a graph of size 4; (b) the tree of the graph in (a) with root 2.

```
4
(0,1)
(0,2)
(0,3)
(2,3)
```

defines a graph shown in Figure **??**(a). You can assume (without checking) that the graph is connected, i.e., there is an undirected path between each pair of nodes in the graph. Ensure that the size of the graph is a positive (non-zero) integer. Self-loops of the form (a, a) are illegal, and an arc must not appear more than once. For example, the following input is illegal:

```
2
(1,2)
(2,1)
```

Graph Data Structures The graph is implemented as a dynamically allocated array, where each element of the array is a structure that describes a node:

```
typedef struct node {
    unsigned int id;                  /* node id */
    Neighbors_list neighbors;         /* nodes directly connected to this node */
    enum {white, gray, black} color; /* data used by BFS algorithm */
    struct node *parent;              /* a neighbor declared as a parent in the tree */
    Children_list children;           /* neighbors declared as children in the tree */
} Node;

typedef struct graph {
  Node * nodes;                       /* graph nodes */
  unsigned int size;                  /* number of nodes in the graph */
} Graph;
```

You need to implement the `Neighbors_list` and `Children_list` data structures as singly-linked lists. The choice of a specific way to implement these singly-linked lists is up to you. Also, you need to implement the functions:

```
void graph_init(Graph * graph, char * file_name);
void graph_clear(Graph * graph);
Node * get_node_by_id(Graph * graph, unsigned int id);
void BFS(Graph * graph, Node * root);
```

The `graph_init()` function reads the graph description from an input file and initializes the graph data-structure. It dynamically allocates the array of the nodes and initializes each node in the array. Then, it inserts arcs as specified in the input file. You may use the provided function `get_line()` to read lines from the input file. The `graph_clear()` clears the graph. It deallocates all internal data-structures, and the function `get_node_by_id()` obtains a graph node given the node id.

BFS Algorithm BFS algorithm systematically explores the arcs of the graph to "discover" all nodes reachable from r. It marks each node with one of the 3 colors *white*, *gray*, or *black*, to keep track of its progress. All nodes are white at the beginning. Whenever a white node b is first discovered by the BFS through an arc (a, b), b is marked with gray, the node a becomes the parent of b in the tree, and b is pushed to the back of the queue. The queue contains nodes that have already been discovered, but have neighbors that are not yet discovered. A node becomes black after all its neighbors have been discovered. At the end of the algorithm, each node has one of its neighbors declared as its parent in the tree, some other neighbors as its children, and all nodes in the graph "agree" on their parent-child relationship. The neighbors of each gray node must be discovered in ascending order of their ids, to guarantee unique results. This can be achieved either by maintaining the list of neighbors sorted, or by sorting them at once during the initialization of the BFS algorithm. You are free to follow one of these two approaches. You need to implement the function `BFS()` in the file `bfs.c`. This function uses a `Queue` data structure, that you need to implement as well. The pseudo-code of the algorithm is listed below.

Output Format The function `tree_print()` provided in `bfs.c` prints out the final tree in the desired format. The children of each node should be printed in ascending order of their ids. This can be achieved either by maintaining the list of children sorted, or by sorting them while they are printed (adding the appropriate code to `tree_print()`.) The tree of the graph in Figure **??**(b) rooted at 2 is described by:

```
(2
 (0
  (1))
 (3))
```

Memory Management You need to implement a memory-management scheme that keeps track of all memory blocks allocated by the program, using a linked-list of allocated memory-blocks. Use `allocate()` to allocate memory blocks and `deallocate()` to deallocate memory blocks in all functions in `bfs.c`. These function are declared in `memory_man.h`. Do not use any other memory allocation and deallocation functions. Make sure that every memory block allocated by your program is deallocated when the block is not needed any longer. Implement the following functions in `memory_man.c`:

- `void * allocate(unsigned int size);`
  Allocates a memory block of the given size using standard `malloc`, adds the block to the list of blocks, and returns the pointer to the block.
- `void deallocate(void * block);`
  If the block is currently allocated, deallocates the given block, and removes it from the list of blocks.

- `void mem_clear(void);`
  Traverses the list of allocated blocks and deallocates all memory blocks currently allocated[2]

If the program terminates abnormally due to an error, the function `mem_clear()`, will deallocate all blocks that are still allocated before it exists the program.

Algorithm 1 BFS - Breath First Search

Require: $G(N, A)$ a graph with nodes N and arcs A
Ensure: G is connected

```
    {Initialization}
 1: for n ∈ N do
 2:    color(n) ← white
 3:    parent(n) ← NULL
 4:    children(n) ← empty_list
 5: end for
 6: Q ← empty_queue
    {Main body}
 7: color(r) ← gray
 8: push_back(Q, r)
 9: while !empty(Q) do
10:    a ← get_top(Q)
11:    pop(Q)
12:    for b ∈ neighbors(a) do
13:       if color(b) == white then
14:          color(b) ← gray
15:          parent(b) ← a
16:          insert(children(a), b)
17:          push_back(Q, b)
18:       end if
19:    end for
20:    color(a) = black
21: end while
```

The operation `get_top()` obtains the element at the top of the queue. The operation `pop()` removes the element at the top of the queue from the queue. The operation `push_back()` inserts an element to the back of the queue. The operation `insert()` inserts an item to the list.

Files Name and Permission to the Files

To make the automatic checker happy, the directory `~/software1/assign4/` should contain the following files: `bfs.h`, `bfs.c`, memory_man.h and memory_man.c. Before submitting the solution set, please give permission to the files by executing the following command:

```
chmod 705 ~ ~/software1 ~/software1/assign4 ~/software1/assign4/*
```

[2]Make sure that the elements of the list are also deallocated.