Programming Assignment 2:

GridWorld

Part I (design): Due Monday Sept 28 at the beginning of class.
you will submit a hard-copy of your design document in class and also an electronic copy via blackboard (which may just be a scan of your writeup).

Part II (implementation): Due Monday Oct. 5 by 1:59PM.

Objectives:

1. Design and implementation of data structures for a given Abstract Data Type.
2. Practice with linked lists
3. Coming up with a design which meets given runtime specifications.

Welcome to GridWorld where everybody is a number and lives in a square (unless of course they are dead).

The world is kind of boring. It is an `RxC` grid (rows and columns). Each entry on the grid is called a *district* and is referred to by its row *i* and column *j* where $i \in \{0..R-1\}$ and $j \in \{0..C-1\}$. We'll sometimes refer to such a district as $D_{i,j}$.

Then there are the people; each person is uniquely identified by an integer. Person IDs start at zero. Each person lives in one (and only one) of the districts.

When a world is created, four things determine its initial configuration:

- `R`: the number of rows
- `C`: the number of columns
- `N`: the number of people initially in the world (numbered 0..N-1).
- `RandFlag`: a boolean flag specifying if the initial population should be assigned to districts at random (flag set to true) or if they should all be initially assigned to district $D_{0,0}$ (flag set to false).

Once a world exists, a number of operations and queries can be performed on it (you should be thinking Abstract Data Type about now). The operations that are supported are given in the table below. These operations are listed in the table below. On the left, are user commands that an

application/tester program will support. On the right (more importantly) are the corresponding ADT functions. Devising an organization of your data to support these operations in the specified runtime is the main focus of the project. Part I will be a written document describing your proposal.

User Command	Corresponding ADT Function (`gw.h`)
none; invoked as part of setup. Num rows, cols, population and rand-flag determined by command line arguments (or default values) After initialization, the command loop starts. (see last section of handout)	**`GW *gw_build(int nrows, int ncols, int pop, int rnd);`** **behavior**: initializes a GridWorld object according to the specifications of the given parameters. **return**: pointer to `gw_struct` (a GW). **runtime**: $O(CR + N)$
members i j **output:** prints list of people living in district (i,j). Order may be arbitrary.	**`int * gw_members(GW *g, int i, int j, int *n);`** **behavior**: returns an integer array containing the members of district (i,j). Also stores in *n the number of entries in the array ($N_{i,j}$) **return**: if (i,j) not valid, NULL is returned; otherwise the array described above is returned. **runtime**: $O(N_{i,j})$

population i j
 	
example: > `population 2 7`
`6`
 	
example: > `population 80 7`
`no such district`
	`int gw_district_pop(GW *g, int i, int j);`
	
	return: the number of people living in district (i,j); if (i,j) is not a valid district, -1 is returned.
	
	runtime: $O(1)$

population **example:** > `population` `10`	**`int gw_total_pop(GW *g);`** **return:** total number of (alive) people in the entire grid. **runtime:** $O(1)$
move x i j **example: >** move 5 2 9	**`int gw_move(GW *g, int x, int i, int j);`** **behavior**: if alive, moves person x from current district to district (i,j).

	return: 1 if succeeds; 0 if fails (no such person). **runtime**: $O(1)$
find x **example:** `> find 8`	`int gw_find(GW *g, int x, int *r, int *c);` **behavior**: if x is alive, and in district (i,j), sets *r to i and *c to j. **return**: 1 if succeeds; 0 if fails (no such person). **runtime**: $O(1)$
kill x **example:** > kill 5	`int gw_kill(GW *g, int x);` **behavior**: if alive, x is removed from it's current district and the entire world. Data structures updated accordingly. **return**: 1 if succeeds; 0 if fails (no such person). **runtime**: $O(1)$
create i j reports person ID (or error message if operation fails.)	`int gw_create(GW *g, int i, int j);` **behavior**: creates a new person, assigns a unique integer ID to the person and puts him/her in district (i,j). It reports to the user the ID of the new person. **return**: integer ID of new person on success; -1 for failure (no such district). **runtime**: $O(1)$ There is a caveat here: this bound will be technically "amortized" because you will be allowed to reallocate an array as more people are created. To achieve the amortized $O(1)$ behavior, your resizing operation should yield an array twice as large as the previous capacity. **details**: when handing out person IDs, you have some freedom in exactly which ID is returned. However, you don't want to just keep giving larger and larger IDs if there are other smaller IDs available (i.e., people have been killed). The reason for this should become apparent as you work on the project.
quit	`void gw_free(GW *g);`

Part I: solution design

In this part you will design a solution to the problem, but will not do any implementation. You should think of this as a preliminary design document.

The document will include the following:

1. An informal description of what it means for an operation to take constant time including examples (not necessarily relating to this project) illustrating constant time and non-constant time operations.
2. Solution description. This should include:
    a. ``boxes and arrows'' diagrams (like we've used in lecture) clearly illustrating how you intend to organize the data.
    b. Descriptions (along with appropriate pseudo-code) of how you will accomplish each operation in the required time. A boxes-and-arrows diagram should make it clear how various data is accessed and modified. The pseudo-code can include English, but must be very clear and described in a step-by-step fashion (not paragraph form!). You will need to make a clear argument that your design meets the runtime requirements. Though you will not be writing real code at this point, you should use meaningful variable names for your main data structures the types of individual data items.

How do you know if your design description is sufficient? Ask yourself this:

> **"Can a competent computer scientist take this document and:**
>
> 1. **be convinced that the requirements for each operation are met and**
> 2. **be able to write an implementation of your design in a language of their choice???"**

Hints and Suggestions:

> **Hint**: doubly-linked lists might be handy!
>
> **Suggestion**: start with a straightforward design that satisfies the *functional* requirements; then analyze the runtime of individual operations and then try to modify the design to meet the requirements.

Your writeup will be submitted as a formatted hardcopy at the beginning of class.

Part II: Implementation

You will submit two files:

`gw.c` (just contains functions and declarations for gw_ operations; it should not have a main function)
`main.c` (simple application program which parses and executes the commands ***as specified in the left hand column of the table above.)***

Behavior of your application (in `main.c`):

The initial configuration of the world will be determined by a set of command-line arguments and flags. If a particular cmd-line argument or flag isn't specified, a default value is assumed.

Suppose your executable is called `gwsim`

```
gwsim [-N <num-people>] [-R <num-rows>] [-C <num-cols>] [-rand]
```

By convention, square brackets are used to indicate that a the argument or switch is optional. We use `<blah-blah>` as a placeholder for a numerical value.

The `-rand` indicates that the people should be assigned at random to districts on initialization.

Default values:

Initial population: 10
Number of rows: 5
Number of columns: 5
Non-randomized initialization (all people assigned to district (0,0)).

Following UNIX conventions, the command line arguments ***can be specified in any order.***

After your program initializes itself, it enters a command-loop asking the user to enter a command (again, the left column of the table above describes how they are supposed to behave).

That's it...