Program 4: Balloon-Pop

In this assignment, you will do the following:

1. Gain more experience with Abstract Data Types and how they can be implemented in C.

2. Experience *implementing* an ADT from a given specification.

3. Using your result from (2) to build a primitive, but playable version of a fun puzzle game. (Actually, a client program playing the game is given to you…).

Very first steps.

1. **Find a Partner:** This assignment may be done in teams of two. In fact, it is encouraged (but ultimately, you may work alone if you prefer).
   If you need us to be a matchmaker, post to piazza and we will try to pair people up.
2. **Play "PopIt":** play the game on this web page:
   http://poppit.pogo.com/hd/PoppitHD.html
   You will be implementing basically this same game with crude character based animation.

3. **Get Started:** Now start reading the rest of this document (hopefully with your partner).
   - Be patient
   - Don't panic
   - Try to have fun -- it's not as hard as it seems at first examination and it will actually be kind of cool when you have a working program.

Due: Thursday, October 21 at 11:59PM.

“Balloon Pop” Overview

You’ve probably played puzzle games a lot like this…

The game starts with a grid of colored balloons. The player selects a balloon to pop; if there is at least one other balloon of the same color “connected” to the selected balloon, the selected balloon does indeed pop along with all other balloons of the same color that are “connected” to it. We will call such a collection a “cluster”.

The balloons are filled with helium, so if a balloon pops, all balloons *below* it rise up (there is a “ceiling” at the top of the grid). This should remind you of the `gravity` exercise from a previous homework.

You can play a version of the game here:

http://poppit.pogo.com/hd/PoppitHD.html

The static screen shots below show a couple of moves.

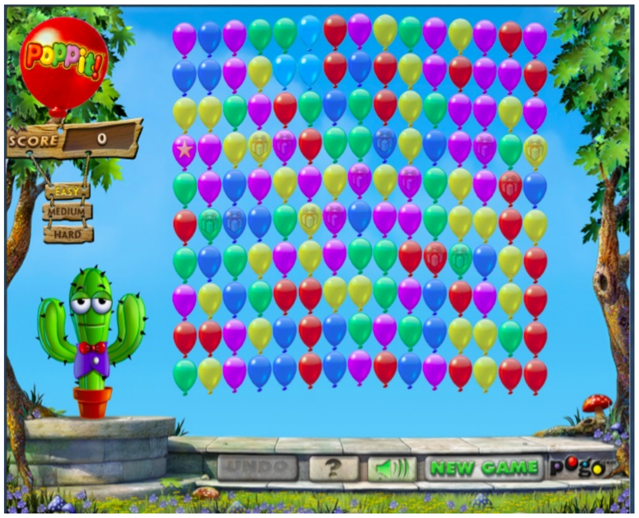
Scoring: for our version, the player is rewarded for popping larger clusters: if a move results in a cluster of size `n` being popped, the player gets `n(n-1)` added to their score.

The game ends when no clusters remain (the end board may or may not have any remaining balloons).

Examples of a couple of moves from an initial board.
The description specifies balloon locations as (row, column) where the upper-left balloon is at (0,0).

Move 1: select green balloon at (4,10)

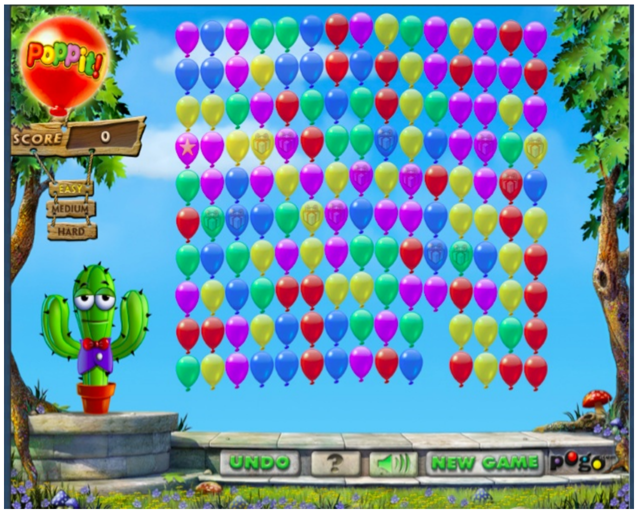
That balloon and its green neighbor below pop.



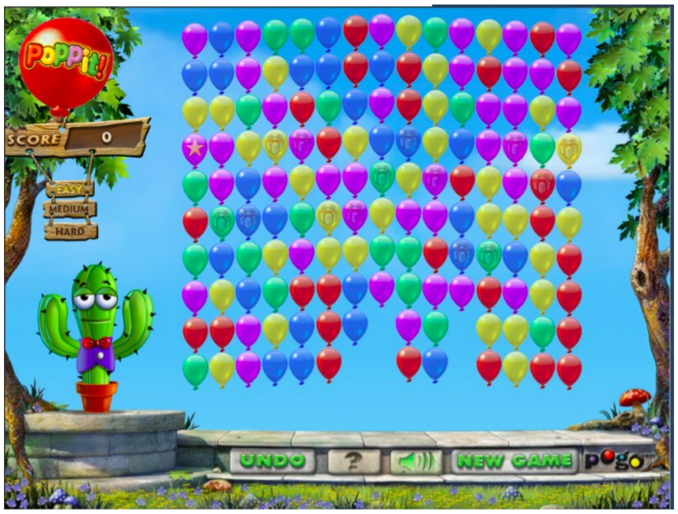
After move 1, balloons below the two popped float up to fill vacant slots.

Move 2: select green balloon (3,6)

A cluster of 3 green balloons is popped



After the three green balloons from Move 2 are popped, lower balloons in the same columns as the popped balloons float up to give the configuration below.



For discussion, consider the purple balloon at (0, 12).

It belongs to a cluster of 8 purple balloons spanning 3 columns and 5 rows.

Your implementation of the game will be accomplished mainly through completing the implementation of an ADT for which I have given you the specifications (in the form of a header file).

Clusters

Let's say clusters of balloons identify groups of friends (bare with me). In the world of balloons we have the following rules to determine the group of friends for a particular balloon `X`:

1. **REFLEXIVITY**: `X` is friends with him/her self.
2. **NEIGHBORS**: If balloon `Y` is a neighbor of `X` (ABOVE, BELOW, LEFT or RIGHT) and has the same color as `X`, then `X` and `Y` are friends.
3. **TRANSITIVITY**: If balloon `Y` is an friend of `X and` balloon `Z` is a friend of `Y,` then `X` and `Z` are also friends.

For completeness, we should also say that these are the ***only*** conditions under which balloons are friends.

So… a cluster is just a set of mutual friends according to the above rules. Some observations:

- Every balloon belongs to exactly one cluster.
- Clusters may be as small as one.
- **The rules sound kind of *recursive* don't they…..? Hint, hint!!!**
- Remember that direct neighbors are above, below, left or right (or N, S, E, W if you prefer). There are no "diagonal" neighbors (though there can of course be diagonal friends).

Creating a Playable Game

You will complete the implementation of an Abstract Data Type representing the state of a balloon-pop game.

The given file `BPGame.h` gives the specification of the ADT. In a nutshell:

Your job is to write a complete implementation of this ADT (in a file called `BPGame.c`).

You are *not* allowed to modify the interface specifications (i.e., `BPGame.h`)

The type `BPGame`: You will notice the following near the top of `BPGame.h`:

```
/**
BPGame is incompletely specified as far as any client program
        is concerned.
    A BPGame "instance" captures everything about a particular run
        of a balloon pop game:  the current state of the board,
        the score, any additional info for the undo feature, etc.

    The definition of actual struct bpgame is hidden in bpgame.c
**/
typedef struct bpgame BPGame;
```

TODO*: You* will have to decide what the structure `BPGame` should contain and specify this in your `bpgame.c` file. The contents of a `BPGame` structure should capture the entire state of a game.

Remember a client program should only be able to operate on a `bpgame` object/instance through the functions specified by the .h file / interface. This is why you are hiding the details in `BPGame.c` -- just like we did with the stack module.

TODO: Implement the functions operating as specified in the `bpgame.h` file. Since you are writing the actual implementation, you do have access to the fields of `BPGame` structures and can dereference `BPGame` pointers.

There are 11 functions to implement. They are listed below, but see `BPGame.h` for detailed specifications of behavior.

`extern BPGame * bb_create(int nrows, int ncols);`
`extern BPGame * bb_create_from_mtx(char mtx[][MAX_COLS],`
`                  int nrows, int ncols);`
`extern void bb_destroy(BPGame * b);`
	Functions for creating, intializing and destroying balloon pop boards

`extern void bb_display(BPGame * b);`
`extern int bb_pop(BPGame * b, int r, int c);`
`extern int bb_is_compact(BPGame * b);`
`extern void bb_float_one_step(BPGame * b);`
`extern int bb_score(BPGame * b);`
`extern int bb_get_balloon(BPGame * b, int r, int c);`
`extern int bb_can_pop(BPGame * b);`
`extern int bb_undo(BPGame * b);`
	`operations/queries supported.`
	
	`bb_pop is where most of the interesting work is done.`
	
	`bb_float_one_step allows primitive animation.`

How To Proceed?

1. Study the function specifications in `bpgame.h`
2. Study them again.
3. Decide how you want to represent a board.
4. Start working on your implementation of the ADT in a file called **`bpgame.c`**
5. Design `struct bpgame.`   Make sure you understand where it belongs and what code has access to the data elements -- use the `stack` ADT from Part 1 as a model.
6. Start working on the functions. Adopt the following approach:
    a. Write a function
    b. Test and debug that function
    c. Move on to next function

What About the "Application"??

A complete and correct implementation of `bpgame.c` gives us most of the primitive operations we need to build a Balloon Pop game. But we still don't have a playable game!

We still don't have a "client" program to manage the user interface, etc. Not to worry!

You have been given a client program **`bpop.c`** which actually plays the game by creating a board (either from a file or at random). It does some very crude animation. You are welcome to modify it to your liking. Or to build an alternative client program from scratch.

You should, of course, test the individual functions via small tester programs first rather than by just trying to run bpop and hoping for the best.

You are also given a `makefile` which you may need to extend depending on your design choices.

Tips/Hints:

Even if you don't get the entire bpop game to work, you can still receive partial credit for correctly implemented individual functions.

Prioritize: for example, the undo operation will be worth no more than 15% of your final score, so you might save it to the end -- you can still get a working game without supporting that operation.

The undo operation should remind you of stacks: you have a stack of game configurations with the current config on top. You should be able to leverage your previous work with stacks (Program-2).

You can write additional functions as helpers (and you should!). They just won't be among the functions client programs can directly call.

You can also add additional structures / typedefs in `bpgame.c`

Submission

Each team will submit a single archive of the directory/folder containing their solution.

There will be one submission per team.

- Decide which team member will make the submission to blackboard.
- Edit the file `team.txt` to list the members of your team.

The key files to include in your submission:

`bpgame.h`
`bpgame.c`
`bpop.c`
`makefile`
`team.txt`
`readme.txt` (if there is any specific explanation you think helps)

Also include any other files source files your implementation relies on (e.g., the `simpleio` files, `stack` files, etc.)

Some sanity checks:

- Make sure that there is no `main` function in `BPGame.c`; if you added one for the purposes of testing, be sure to remove it or comment it out before submission.
- Your `makefile` should support the followings:

```
make bpgame.o

make bpop
```

If your code depends on other files (e.g., a `stack` module) your makefile must automatically compile and link such files only by the above runs of `make`.

In other words, the graders should not need to do anything besides the two make commands above to compile your code.

- Finally, zip your file and just upload the zip file, “`team-netIDs.zip`”, to gradescope.