CMSC 216 Project #6 Spring 2016

Bag and testing program **Deadline: Tuesday April 26, 8:00PM**

In this project you will use dynamic memory allocation to implement a data structure called a *bag* (also called a *multiset*). You will also write a black–box testing program to test the functionality of implementations of the bag that we will provide, which may or may not be correct. Most of the secret tests for this project will check your testing program.

Like a set, a bag stores a collection of elements without any specific ordering, but the difference between a set and a bag is that a bag can contain multiple occurrences of elements. For example, suppose a bag is storing character strings (which will be the case in this project), and the elements "platypus", "koala", "quokka", and "koala" are added to it– it will contain one occurrence of both elements "platypus" and "quokka", and two occurrences of "koala". If an occurrence of "koala" is removed, the bag will still contain one occurrence of it. Only if both occurrences are removed will "koala" not be present in the bag at all.

You **must** use dynamic memory allocation to implement a bag. There is **no limit** to the number of elements that a bag should be able to store, and there is also **no limit** to the length (number of characters) of any particular element that can be stored in a bag. You can use any form of dynamically–allocated memory to implement bags, for example dynamically–allocated arrays, binary search trees, linked lists, etc. You can also reuse code that you wrote in earlier projects for storing bags, if desired.

1 Bag functions to be written, and `Makefile`

1.1 Bag functions

Since many operations in this project require dynamic memory allocation, and memory allocation calls can fail, you will need to check the results of every memory allocation call to ensure your code is executing correctly. Non–`void` functions where memory allocation fails should return -1.

We are not giving you any type declarations for this project; the exact way that you implement bags is up to you. The file `bag.h` only contains two definitions mentioned below, and the prototypes of the functions that you have to write. Instead of type definitions it includes a header file `bag-implementation.h`, which is not provided. You will write this file, which must contain a definition of the type `Bag` that the functions operate upon, and any other types that your functions need.

`void init_bag(Bag *bag)`**:** This function should initialize the bag whose memory address is passed in as a parameter. It must be called once (and only once) on a `Bag` variable, before elements can be stored in it. Exactly what it does depends on how you decide to implement bags. If the parameter `bag` is `NULL`, the function should have no effect.

All of the other functions may assume that `init_bag()` has already been called on any bag variables that are passed into them.

`void add_to_bag(Bag *bag, const char *element)`**:** This function should add one occurrence of the element `element` to the bag that its parameter `bag` points to. If `element` was not previously in the bag at all, it will subsequently be present with one occurrence. If it was in the bag, it will have one more occurrence than it had before. If either `bag` or `element` are `NULL`, the function should just return without changing anything.

Your function should make a copy of the string passed into `element`, and store the copy in the bag.

`size_t size(Bag bag)`**:** This function should return the number of **elements** currently stored in its parameter `bag`, ignoring the number of occurrences of the elements. For example, if called on a bag storing one occurrence of "koala" and two occurrences of "platypus", 2 should be returned. The parameter `bag` should not be modified.

`int count(Bag bag, const char *element)`**:** This function should return the number of **occurrences** of the element `element` that are currently stored in its parameter `bag`. (The total number of elements stored

doesn't matter, nor does the occurrences of other elements.) If `element` is not present in the bag at all, the function should return -1. If `element` is `NULL`, the function should also return -1. The parameter `bag` should not be modified.

For example, suppose a bag is storing one occurrence of "platypus", two occurrences of "koala", and one occurrence of "quokka". Then calling this function on that bag and "koala" should return 2. Calling it on "platypus" should return 1, and calling it on "elephant" should return -1.

Note that the function should never return 0; if an element is present, it will have one or more occurrences, and if it isn't present at all, -1 will be returned.

`int remove_occurrence(Bag *bag, const char *element)`: This function should remove one occurrence of the element `element` from the bag that its parameter `bag` points to. The number of occurrences of that element should subsequently decrease by 1. If `element` is not present in the bag at all, the function should return -1. If it is present, an occurrence of it should be removed, and the function should return the **new** number of occurrences that it has, **except** if its last or only occurrence is being removed, in which case -1 should be returned. If either `bag` or `element` are `NULL`, the function should also return -1 without changing anything.

Note that the function should never return 0; if an element is present with more than one occurrence it will still have one or more occurrences after one occurrence is removed, while if it is present only once then -1 will be returned when that one occurrence is removed, and if it isn't present at all, -1 will also be returned.

`int remove_from_bag(Bag *bag, const char *element)`: This function should remove **all** occurrences of `element` from the bag that its parameter `bag` points to. The size of the bag should subsequently decrease by 1. If `element` is in the bag, the function should remove it and return 0, otherwise, it should just return -1 without changing anything. If either `bag` or `element` are `NULL`, the function should also return -1 without changing anything.

`Bag bag_union(Bag bag1, Bag bag2)`: This function should create and return a new bag which has all of the elements of its two parameter bags `bag1` and `bag2` combined. If an element is only in one of the parameter bags, it will be in the returned bag with the same number of occurrences as it has in that parameter bag. If an element is in both parameter bags, it should be in the returned bag with the combined number of occurrences that it has in both parameter bags. The two parameter bags should not be modified.

Your function should make copies of the strings in its two bag parameters and store those copies in the new bag to be returned.

`int is_sub_bag(Bag bag1, Bag bag2)`: This function should test whether every element in `bag1` is present in `bag2` with at least as many occurrences as it has in `bag1` (meaning the same or more), and return 1 if so. If any element is in `bag1` but not in `bag2`, or if any element is in both bags but it has more occurrences in `bag1` than it does in `bag2`, the function should return 0. Note that if `bag1` has no elements, the function should always return 1, regardless of how many elements `bag2` has. The two parameter bags should not be modified.

`void clear_bag(Bag *bag)`: This function should free **all** dynamically–allocated memory associated with the bag that its parameter `bag` points to. If the parameter `bag` is `NULL`, it should have no effect.

Notes:

- It is invalid to call any function other than `init_bag()` on a bag after `clear_bag()` is called on it.
- It's only valid to call `init_bag()` on a newly–declared bag, and immediately after `clear_bag()` is called on it (before any other function is called on it).

1.2 Makefile

You will have to write a `Makefile` that compiles your code on the public tests. Your `Makefile` should be set up so that all programs are built using separate compilation (i.e., source files are turned into object files, which are then linked together in a separate step). All object files should be built using the following options:


```
-ansi -Wall -g -O0 -Wwrite-strings -Wshadow -pedantic-errors -fstack-protector-all
```

You will need to have the following targets present in your `Makefile`:

1. `all`: make all executables
2. `public01`, `public02`, `public03`, and `public04`: each one building a public test
3. `bag.o`:
4. `my_memory_checker_216.o`: this is used by the last public test
5. `clean`: delete all object files and executables

While you are welcome to add other targets to your `Makefile`, the ones listed above must be present. Make sure you add all the necessary targets that avoid any unnecessary compilation (hint: use the touch command to check your makefile).

2 Writing a program to test our implementations

You will also write a program that determines which of several implementations of the bag data structure that we created contain bugs, and which ones are correct. You will have to write a variety of tests (all contained in one single testing program) that can determine this, based upon the behavior that the functions are described to have in Section 1.1 above. This will be an example of black–box testing, since you will not see the source code of the implementations that your testing program is testing (they will be secret tests).

For these secret tests we will be creating several different implementations of the bag functions described by the prototypes in `bag.h`, some correct and others with errors. Along with your functions in `bag.c` you need to write a program in the file `tester.c` that will test these implementations. `tester.c` should include `bag.h` but not any other files, since we will compile it with our makefile. In particular, it should **not** include `bag.c` (source files normally never `#include` other source files), since we will be linking it with different implementations of the functions for grading. Ideally you would be able to write different tests in separate source files, but due to difficulties in setting things up on the submit server, all of your tests of our bag implementations will have to be in the same single file `tester.c`.

Your submission **must contain** a file named `tester.c` that successfully compiles, or your entire project submission will fail to compile for any tests when submitted. It doesn't matter what your `tester.c` program does (although if it doesn't do anything you will lose the credit for all of the secret tests that use it), but even if you haven't started writing your testing program yet your submission **must** at least contain a compilable `tester.c` file in order to work for any tests. The code supplied to you includes an example `tester.c` file that you can add to, to create your own actual `tester.c`. The provided `tester.c` only tests two things– whether `size()` returns the right value for a bag with several elements, and that `count()` returns the right value for an element that has several occurrences in a bag. Since all of the bag tests have to be in the single source file `tester.c`, we put each of these tests in a separate function in that file, just to organize things cleanly. You are encouraged to write separate functions for each thing your tester is testing also.

Your tester should exercise our different implementations in such a way that you test for and detect as many error conditions as possible, given that you won't be able to see the code that implements them. Your score on these secret tests will be based on how well your tests detect the correct versus the incorrect implementations. If your testing program does not detect any errors in the functions it is calling it must quit with an exit status of `CORRECT` (which is defined in `bag.h`). If your program detects any errors in the functions it is calling it must quit with an exit status of `FOUND_BUG` (also defined in `bag.h`). (Note that both of these are defined to be atypical exit status values.) It doesn't matter what output your program produces before quitting; you may have it produce whatever output, if any, you find useful.

Note that the secret tests that check your testing program will be "backwards" in some sense compared to the other public and secret tests of this and other projects, in that we usually write the main programs that are used as tests, which call functions that you write. However, with the tests of your testing program, your `tester.c` is the main program, and our implementations contain the functions it will call. Keep this in mind when writing makefile rules for your own tests of this part of the project.

You will have to carefully read the descriptions of the bag functions in Section 1.1 above, identify as many ways you can think of that the functions could have incorrect behavior, and write tests in your `tester.c` that would check for that behavior. Then, to test your `tester.c`, make copies of your `bag.c` (for example, `bag-wrong1.c`, `bag-wrong2.c`, etc.), introduce bugs into them, and ensure that your testing program finds the bugs (i.e., that it exits with `FOUND_BUG` when compiled and linked with each buggy version you created). Also make sure that your tester continues to exit with `CORRECT` when linked with your correct bag implementation in `bag.c`.

The constraints on what your testing program may and may not do are:

- Our functions do not have any bugs that would cause program crashes (assuming you pass valid pointers into the functions that have pointers as parameters). All the bugs will involve functions returning the wrong value, or not storing information into a bag correctly according to the descriptions of their behavior above.
- Bugs in our functions will not include memory leaks or memory corruptions, so you do not need to test for these kinds of problems. Furthermore, our `Makefile` that is compiling and linking your tester with the secret tests is assuming that your tester is not using any memory checking routines, so not only do our functions not have them, you shouldn't even attempt to test for these kinds of errors in our implementations, because your tester won't even compile on the submit server if you do.
- Your testing code will only be able to do black–box testing. In particular, you will not be able to examine or modify the internals of bags in testing our implementations, because you have no idea what data structures we're using to store bags. You'll only be able to test our implementations by calling the functions given in `bag.h`. In other words, your tester cannot assume any representation for bags. Your tester is going to be compiled with your bag, **and** with our bags. It **cannot** assume that our bags are implemented the same way as yours, and try to access any of the internal structure of a bag so your tester may compile and run fine for you, but this will not compile with our bag implementations.
- Your tester can **only** call the functions in `bag.h` on bags. If you try to call helper functions that you wrote on bags these will be not be the same in our implementations of bags as in yours, and your tester will not compile on the submit server.
- You may assume that memory allocations performed by our bag versions will always succeed.

Some hints regarding your testing program:

- Read the descriptions of the functions above very carefully– several times– because their behavior as described is what your testing program needs to test for.
- Be sure to test your testing program with a correct bag implementation. In this case `CORRECT` should be returned. Don't test only with incorrect bag implementation, and overlook this simple case.
- You are welcome to use `lcov` to aid in writing your testing code, so that you can be sure that your tests at least cover all the parts of **your** function implementations. However, if you do, you must **remove** all `lcov`–related options from your `Makefile` rules before submitting your project, otherwise, your code may not compile on the submit server. It will be much easier to remove the `lcov` options if you add a variable which has the options (such as `LCOVFLAGS`) at the top of your `Makefile`, which you can just change in one place before submitting.

Note that you may not give any parts **or ideas** from your project to anyone else, **including test data**. This means that not only may you not give anyone else your `tester.c`, you may also not provide them with any ideas or suggestions of how to write it or what to test in it.

3 Submission and grading criteria

You can submit your project by executing the **submit** command in your project directory. We will use our versions of all the header files that we are supplying to build our tests, so do not make any changes to the header files. (Of course we will use your version of `bag-implementation.h`.)

If you try to submit your project in grace, and you get the error:

"Exception in thread 'main' java.lang.OutOfMemoryError: unable to create new native thread"

then close all terminal windows except one, and try to submit again.

Your project grade will be determined with the following weights:

Results of public tests	10%
Results of secret tests	80%
Style	10%

4 Academic Integrity

Please see the syllabus for project rules and academic integrity information. All programming assignments in this course are to be written individually (unless explicitly indicated otherwise in a written project handout). Cooperation between students is a violation of the Code of Academic Integrity.

