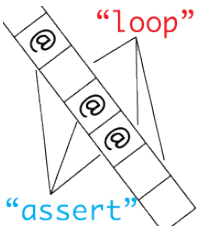
15-122: Principles of Imperative Computation, Spring 2025

Programming Homework 7: Bloom Filters



Due: Sunday 16th March, 2025 by 9pm EDT

In this assignment, you will implement a variation of hash-table-based sets, called *Bloom filters*, and explore some of the applications of Bloom filters. Along the way, you will build a pipeline of C0 and C1 libraries and experiment with a space-efficient representation.

From your `private/15122` directory, run

```
% autolab122 download bloom
```

to download the code handout for this assignment (you can also find it on Autolab). The file `README.txt` in the code handout goes over the contents of the handout and explains how to hand the assignment in. There is a TEN (10) PENALTY-FREE HANDIN LIMIT. Every additional handin will incur a small (5%) penalty (even if using a late day). Your score for this assignment will be the score of your last Autolab submission.

To help you write test cases for Task 1, which we want you to do before starting on the later tasks, there will be a separate, unofficial "Bloom Filter Test Case Checker" Autolab assignment. This will only run the part of the autograder that checks Task 1. There is no handin limit for that unofficial autograder. Check the `README.txt` file carefully for details.

Important Tasks 2–5 of this assignment ask you to work with *function pointers*. In some semesters, function pointers are covered in lecture a day or two after this assignment is released — but always long before it is due. You may either wait until then, or proactively read the relevant course material. Tasks 1, 6 and 7 do not rely on function pointers and can be completed right away.

1 Bloom Filters

Fundamentally, Bloom filters are an implementation of the *set interface* discussed in class.

```
// typedef ______* bloom_t;

bloom_t bloom_new(int capacity)
  /*@requires 0 < capacity; @*/
  /*@ensures \result != NULL; @*/ ;

bool bloom_contains(bloom_t B, string x)
  /*@requires B != NULL; @*/ ;

void bloom_add(bloom_t B, string x)
  /*@requires B != NULL; @*/
  /*@ensures bloom_contains(B, x); @*/ ;
```

The one interesting twist is that the `bloom_contains` function may lie: if the Bloom filter says that something *is* in the set, it may be wrong — this is called a *false positive*; but if it says that something *is not* in the set, it is always right. To put it a different way, a Bloom filter answers the question "is `x` in the set?" with either "no" or "maybe".

Here's a summary of how the output of `bloom_contains(B,x)` may relate to whether `x` is in `B`. The words in all-caps describe when each combination happens. As you can see, false positives (**SOMETIMES!?**) are problematic: `x` may not be in `B`, and yet `bloom_contains(B,x)` returns `true`.

if ↓ *then* →	`x` is in `B`	`x` is **not** in `B`
`bloom_contains(B,x)` returns `true`	**SOMETIMES** *(true positive)*	**SOMETIMES!?** *(false positive)*
`bloom_contains(B,x)` returns `false`	**NEVER** *(false negative)*	**ALWAYS** *(true negative)*

Different implementations of the Bloom filter interface may behave in radically different ways, and yet be correct according to this description:

- One such implementation could *always* returns `true`, signaling "maybe `x` is in the set" for all `x`. You've been provided with this implementation in `bloom-worst.o0`. It is very fast and uses very little space! Although correct, it is totally useless to a client.
- At the other end of the spectrum, note that a proper set is a Bloom filter that never returns false positives. Thus one can implement the Bloom filter operations so that it behaves like a set. You have been provided with one such implementation in `bloom-expensive.o0`. It only signals "maybe `x` is in the set" when `x` is *really, actually* in the set. This is obviously correct, but it uses lots of memory, and what's more, it is quite slow.

The implementations we're going to explore in this assignment will be in-between: they use less memory than a real hash set, but give fewer false positives than a completely non-committal implementation. But first, let's write some tests!

1.1 Testing Bloom Filters

Our tests will be on Bloom filters whose elements are strings. Since we will also be using other element types later, we prefix the interface type and functions with "`string_`". You can find the resulting interface on page 11 of this writeup.

Task 1 (6 points) In file `string-test.c0`, write a testing program that respects the interface for Bloom filters with **`string`** elements on page 11. It should serve two purposes:

- The testing program should attempt to raise an assertion error on any incorrect implementation of the interface (e.g., if it were to return a false negative).
- On any correct implementation of the Bloom filter interface, the `main` function should return an *accuracy score* from 0 and 100 (inclusive).
  - On the worst possible Bloom filter implementation described above, the accuracy score should be 0.
  - On an "error free" Bloom filter implementation (such as an actual set), the accuracy score should be 100.
  - On any Bloom filter implementation that has some false positives and some (true) negatives, the accuracy score should be between 0 and 100.

We ask that you do not call `string_bloom_new` with a capacity greater than or equal to `int_max()/8`.

Generally speaking, worse implementations should have lower accuracy scores. You will be graded in part based on whether your tests are able to distinguish relatively bad (but not fully non-committal) implementations from relatively good (but not perfect) ones.

An idea to keep in mind when you are writing your tests is that Bloom filters, like hash tables, have a *load factor*. If n is the total number of distinct elements that have been inserted and m is the input to `string_bloom_new(`m`)`, then the load factor is n/m. We will generally expect lots of false positives when the load factor exceeds 1, and vastly fewer false positives when the load factor is much smaller than 1.

You are strongly encouraged to go ahead and submit to Autolab using the unofficial "Bloom Filter Test Case Checker" autograder before you move on from this task.

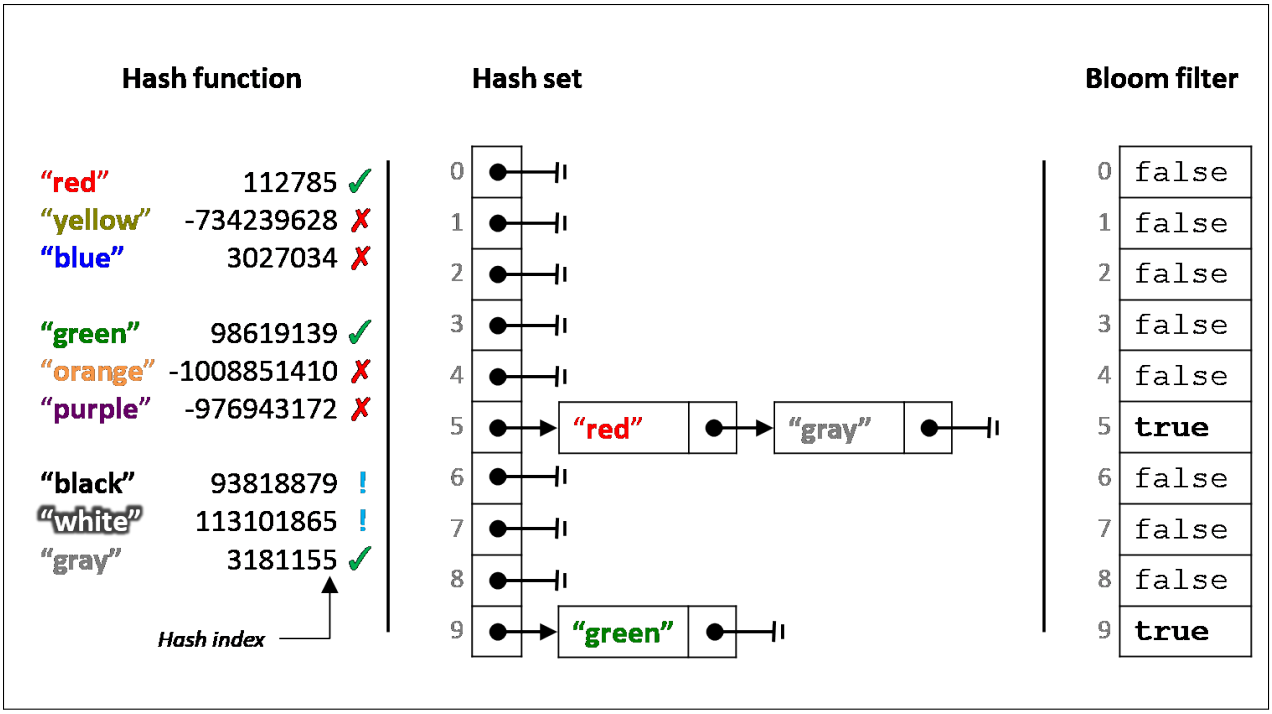


Figure 1: On the left, some strings and their hash values according to the `hash_mul31` hash function. In the middle, a separate-chaining hash table using this hash function, after the insertion of the strings `"green"`, `"red"`, and `"gray"`. Next to them is an indication of whether the Bloom filter (first implementation) yields a true positive (✔), a false positive (!), or a (true) negative (✘). On the right, a Bloom filter using this hash function, after the insertion of the strings `"green"`, `"red"`, and `"gray"`.

2 Hash-based Bloom Filters

Bloom filters are typically implemented by borrowing ideas from hash tables, with some twists.

2.1 Look Ma, no Chains!

The basic idea underlying our implementations of Bloom filters will be to take a regular separate-chaining hash table and get rid of the chains. Instead of the hash table being an array of chain pointers, it will be an array of Booleans.[1] Each index in the Boolean array is `false` if the corresponding hash table bucket is empty, and `true` if the corresponding hash table bucket is non-empty.

In the string-based example in Figure 1, we would give false positives for `"white"` and `"black"`, since these strings collide with strings that are in the set. We would correctly return `false` when asked if other strings, like `"yellow"`, are in the set.

Because Bloom filters do not have chains, they use much less space than an actual hash set. The trade-off, however, is decreased accuracy.

[1] Our implementation will be a bit more abstract than this. You'll see in the next section.

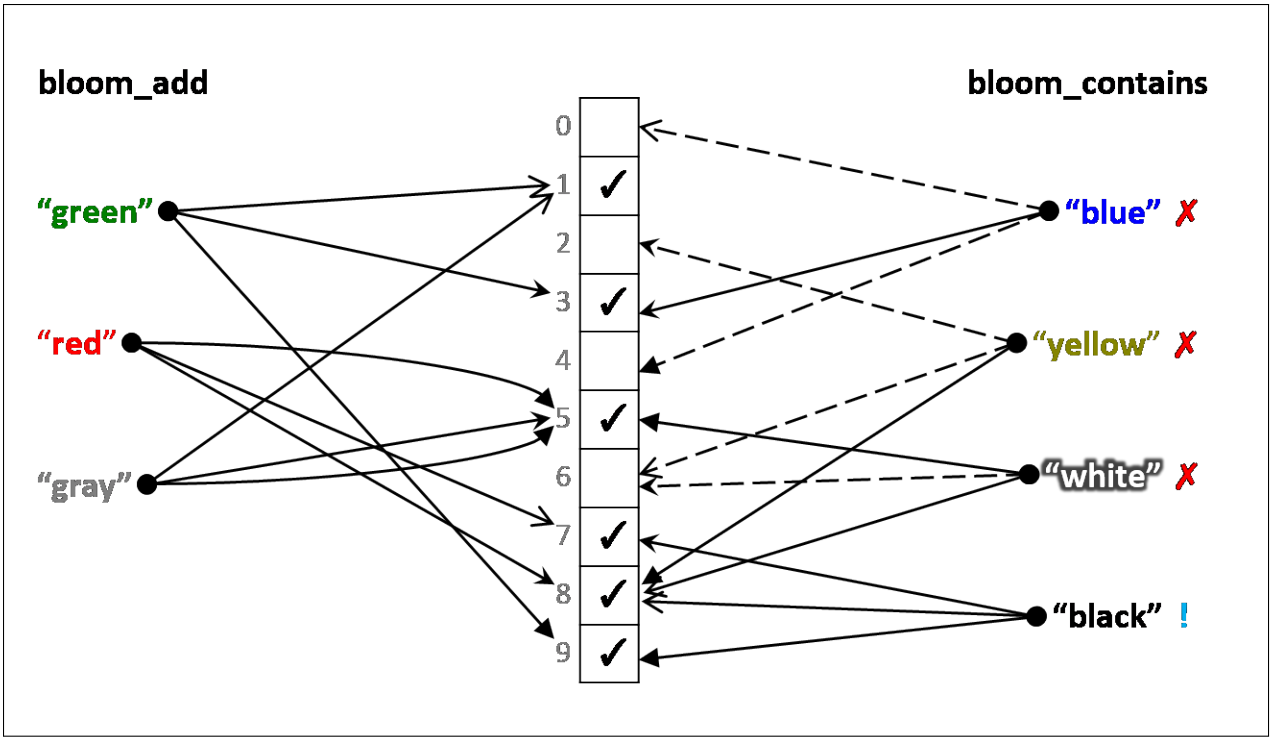


Figure 2: A Bloom filter using multiple hash functions.

2.2 Multiple Hash Functions

It's inevitable to have collisions in a hash table; we tolerate these collisions because they only make the operations a little bit slower. In Bloom filters, however, collisions cause us to get false positives, i.e., loss of accuracy. It's worth going to greater lengths to avoid this.

Increasing m, the size of the table, will help some. However, this strategy only takes us so far. Another remarkably effective strategy is implementing *multiple* hash functions, and inserting each element with *every* available hash function. This means that more of the hash table gets filled up with `true` values (represented as checkmarks in Figure 2). When we test whether an element is in the hash table, then we check all of the indices where that element should hash. If any of them are `false`, we can conclude that the element was never added to the hash table.

The mathematics of why this works better than just growing the table will be a topic for future courses (including Computational Discrete Mathematics, Probability and Computing, and Algorithms). Looking at Figure 2, we see the result of putting our example strings into a Bloom filter. In that figure, the three hash indices we pick are simply the last three digits of the `hash_mul31` hash values that we saw in Figure 1. In an actual implementation, we will want to use three entirely different hash functions.

While we still have a false positive for the string `"black"`, the Bloom filter has fewer false positives than before.

3 Implementing Bloom Filters

In the previous section, we gave examples of Bloom filters whose elements were strings. We will now implement a general framework that allows us to produce Bloom filters for any element type. Later, you'll use this generic library to create specialized Bloom filter libraries — ones that provide functions that operate directly on elements of particular concrete types (like **string**).

3.1 A Bloom Filter Factory

Before we can add elements to and query a Bloom filter, we need to set it up. We do so by creating the underlying table and equipping it with hash functions. It would be terrible if we could add a hash function to a Bloom filter that already contains elements: `bloom_contains` may return `false` on them if this new hash function takes us to a table index containing `false` — we would have false negatives.

We avoid this by defining a *Bloom maker* library. This library provides the following functions that will allow us to create and use the specific Bloom filters discussed earlier.

- `bloommaker_new` creates a Bloom maker of a given capacity, but with no hash functions.
- `bloommaker_addhash` adds a hash function to a Bloom maker.
- `bloommaker_freeze` freezes a Bloom maker, so that no more hash functions can be added to it. At this point, we will be able to use the Bloom maker as a generic Bloom filter.
- `bloommaker_frozen` checks whether a Bloom maker was frozen with `bloommaker_freeze`.
- `bloommaker_contains` implements `bloom_contains` on a frozen Bloom maker.
- `bloommaker_add` implements `bloom_add` on a frozen Bloom maker.

The interface of Bloom makers, complete with contracts, can be found on page 12 of this writeup.

In the library implementation, Bloom makers have the following concrete type:

```
struct bmk_header {
  int capacity;    // capacity > 0
  bt_t table;      // bt_size(table) == capacity
  list* hashes;    // NULL-terminated
  bool frozen;
};
```

where `capacity` is the size of the Boolean table, the table itself has abstract type `bt_t` whose interface is defined on page 13 of this writeup (more on this later), `hashes` is a `NULL`-terminated linked list of hash functions, and `frozen` is a flag that is `true` to indicate that the Bloom maker has been frozen, and `false` otherwise.

Task 2 (8 points) In file `bloommaker.c1`, implement the interface of the Bloom maker library according to this description. Map hash values to table indices as implemented in class. You must write and correctly use a data structure invariant `is_bmk(B)`.

Testing your Bloom maker implementation will be easier once you complete the next task. For now, you may want to just check that it compiles by creating the file `my-bmk-test.c0` containing a trivial `main` function and running the following command:

```
% cc0 -d bt.o0 bloommaker.c1 my-bmk-test.c0
```

Of course, your implementation will need to be correct to get full points.

3.2 Many Blooms

Next, you will use your Bloom maker to implement three concrete Bloom filters. To do so, you will need to commit to specific element types and provide appropriate hash functions.

Task 3 (3 points) In a new file `string-duo.c1`, implement a Bloom filter with elements of type **`string`** that uses mult-31 and LCG as its hash functions. Your code must have the same interface, displayed on page 11, as your tests in Task 1 — in particular you will need to define the type `string_bloom_t`. Note that its elements are strings, *not string pointers*.

The file `hashes.c0` contains the C0 implementation of these and other string-based hash functions (in case you want to build other Bloom filters). Copy `hash_mul31` and `hash_lcg` into `string-duo.c1` and adapt them so that you can use them with your Bloom maker.

The best way to test your code (and your Bloom maker) is by running it against your test cases from Task 1. This may even lead you to improve these test cases.

Task 4 (1 point) In a new file `string-worst.c1`, use your Bloom maker to implement a Bloom filter with elements of type **`string`** that always returns `true`. It should behave just like `bloom-worst.o0`.

Next, let's write a Bloom filter whose elements are not strings, but *meetings*. Meetings are defined to have the following type:

```
struct meeting_header {
  string date;   // e.g. "2pm on 2/21/2025"
  int duration;  // >= 0
};
typedef struct meeting_header meeting;
```

A Bloom filter with elements of this type could be the starting point of a application to book meetings: use `bloom_contains` to check whether a slot is definitely available, and if so book it with `bloom_add`.

Task 5 (2 points) In file `meeting-lcgplus.c1`, use your Bloom maker to implement a Bloom filter whose elements have type `meeting*` and that uses `hash_meeting` as its hash function. A C0 version of this function has been written for you in this file, but like earlier you will need to adapt it to work with your Bloom maker. The interface of this Bloom filter is displayed on page 11. You should treat the `date` field as an arbitrary string. You should create a test file similar to `string-test.c0` but for meetings to test your code.

3.3 Sizing Down

Our Bloom filters are more space-efficient than an actual hash set since there are no chains. Implementing the table itself (the abstract type `bt_t`) as an array of Booleans saves additional space: in C0, a pointer is 64 bits long while a **`bool`** takes up just 8 bits. That's saving of 7 eighths, or 87.5%.

We can do even better however. An individual bit can have values 1 and 0, which we can use to represent "true" and "false". Thus, if we implement the table of a Bloom filter as an array of bits, we can get a saving of 63 sixty fourths, well over 98%, over a hash set table.

Bits are not individually addressable in C0. But an integer consists of 32 bits, each of which can be manipulated using bitwise operations and shifts. Therefore, we can store 32 true/false values in an **`int`**. To store m true/false values, we will need $\lceil m/32 \rceil$ integers (some bits may go unused).

Task 6 (3 points) In file `bt-bit.c0`, complete the implementation of the Boolean table interface on page 13 of this writeup using the provided definition for the type `bt`. You must write and correctly use a data structure invariant `is_bt(T)`.

This implementation of the Boolean table interface should behave exactly like the implementation in `bt.o0`. You can use coin to make sure it does.

Once you are confident it works as expected, you can run your test cases for the earlier tasks by replacing `bt.o0` with `bt-bit.c0` on each compilation command. They should behave identically, but will use a lot less space.

4 An Application of Bloom Filters

Bloom filters come in handy in applications where one needs to check whether a value v is present in a given set S but this check is expensive. Here's the idea: first check v against a Bloom filter into which the elements of S have been inserted; if this quick preliminary check returns `false`, we know with confidence that v is not in S; if instead it returns `true`, we need to check v against S itself, which is expensive.

One such application is making sure that a user does not choose a password that is known to have been compromised. To this end, we will develop a library with the following interface:

```
// typedef ______* pwDB_t;

pwDB_t pwDB_new(int capacity, string badpwfile)
/*@requires capacity > 0 && string_length(badpwfile) > 0;    @*/
/*@ensures \result != NULL;                                  @*/ ;

int pwDB_check(pwDB_t B, string s)
/*@requires B != NULL && string_length(s) > 0;               @*/
/*@ensures 0 <= \result && \result <= 2;                     @*/ ;
```

The function `pwDB_new` populates a new password database (of type `pwDB_t`) with compromised passwords from file `badpwfile`. This bad password database uses a Bloom filter of capacity `capacity` to which the passwords in `badpwfile` have been added.

The function `pwDB_check` checks whether `s` is a compromised password, but it does so in a smart way. It returns `2` if `s` is uncompromised and a quick check in the Bloom filter was sufficient to conclude this, `1` if it is uncompromised but checking the bad password file was necessary to establish this, and `0` if it is compromised. In an actual password setting application, the function `pwDB_check` would be called each time a user attempts to set a new password: if the password is found to be compromised, it would be rejected and the user would be prompted to choose a different password.

The concrete type `pwDB` of bad password databases is defined as follows:

```
struct pwDB_header {
  string_bloom_t filter; // != NULL
  string badpwfile;    // string_length(badpwfile) > 0; badpwfile exists
};
typedef struct pwDB_header pwDB;
```

Besides the underlying Bloom filter (field `filter`), it records the name (`badpwfile`) of the compromised password file. You may not modify this type.

Task 7 (2 points) Complete the implementation of the above interface in file `pw.c0`. Include the data structure invariant function `is_pwDB` and use it in your code. You will need to implement the helper function `pwDB_thoroughcheck` that checks if a word is in the bad password file. You will need to familiarize yourself with the functions in the `<file>` library to write your code.

You may use the file `test-pw.c0` to run some tests. The file `data/20-badpw.txt` contains the 20 most common passwords of 2021 (and yes, `"password"` is one of them). Feel free to use other common password files or to make up your own. Each password should be on a separate line.

4.1 What Else are Bloom Filters Good for?

Screening for compromised passwords is a neat application of Bloom filters. They have many other uses.

Fast First Passes Making sure that a user doesn't pick a password that is known to be compromised uses Bloom filters to quickly check that an action is definitely permitted, falling back to a much more expensive check if this fast first pass says it may not be allowed. There are other applications that fall in this category.

If you've ever used a full-featured text editor like Microsoft Word, you've probably had the experience of watching as the spell checker highlights all the misspelled words in a document. A Bloom filter speeds up this process by storing all the correctly-spelled words. On the first pass, Word reports misspellings only when the Bloom filter says a word definitely isn't spelled correctly. Then the maybe-correctly-spelled words can be checked in a second pass to weed out the false positives. In this case, false positives would be incorrectly-spelled words that the Bloom filter did not flag as misspelled.

Simple Rules, Expensive Exceptions When dealing with human concepts like language, maps, traffic law, or time zones, it's sometimes possible to write a simple algorithm that *usually* gives the right answer. However, these simple algorithms almost always have to be augmented with a large database containing the idiosyncratic exceptions. A Bloom filter can record all the places where our simplistic algorithm *doesn't* return the right answer. Then, we can quickly ask the Bloom filter "is this one of the exceptions where the simple algorithm doesn't work?" If the answer is "no," we use our simple algorithm. If the answer is "maybe," then we look it up in our carefully-maintained database.

Human language was one of the original motivating examples for Bloom filters.[2] Burton Bloom imagined an extensive database of rules for hyphenating English words in a text editor. A Bloom filter captured all the words that can't be hyphenated automatically with a simple algorithm, requiring a database lookup.

One-Hit Wonders Wikipedia describes several additional use cases for Bloom filters.[3] Services like Akamai or Netflix try to store copies of frequently-used content physically close to users, but they have limited storage space.

Due to the way people use such services, a good rule in practice is that, if two people in the same region request the same content, it's worth storing a copy of that content near them. This avoids copying "one-hit wonders", content that only one person wants to view.

A Bloom filter can help with this problem by storing all the recent content requests for each area. Whenever new content is requested, the Bloom filter is asked whether anybody else nearby recently requested that material. If the Bloom filter says "maybe," then the server assumes this is the second request and copies it if not already there. False positives mean that some one-hit wonders get stored, but the trade-off is sometimes worth it.

[2] Bloom, Burton H. "Space/time trade-offs in hash coding with allowable errors." Communications of the ACM, Volume 13 Issue 7, July 1970.

[3] https://en.wikipedia.org/wiki/Bloom_filter

A REFERENCE: Library Interfaces

A.1 Bloom Filters with **string** Elements

```
/****************************** Interface *******************************/

// typedef ______* string_bloom_t;

string_bloom_t string_bloom_new(int capacity)
  /*@requires 0 < capacity; @*/
  /*@ensures \result != NULL; @*/ ;

bool string_bloom_contains(string_bloom_t B, string x)
  /*@requires B != NULL; @*/ ;

void string_bloom_add(string_bloom_t B, string x)
  /*@requires B != NULL; @*/
  /*@ensures string_bloom_contains(B, x); @*/ ;
```

You can also display this interface by running the terminal command

```
% cc0 -i bloom-worst.o0
```

A.2 Bloom Filters with **meeting*** Elements

```
/****************************** Interface *******************************/

// typedef ______* meeting_bloom_t;

meeting_bloom_t meeting_bloom_new(int capacity)
  /*@requires 0 < capacity; @*/
  /*@ensures \result != NULL; @*/ ;

bool meeting_bloom_contains(meeting_bloom_t B, meeting* x)
  /*@requires B != NULL; @*/ ;

void meeting_bloom_add(meeting_bloom_t B, meeting* x)
  /*@requires B != NULL; @*/
  /*@ensures meeting_bloom_contains(B, x); @*/ ;
```

A.3 Generic Bloom Makers

```
/*************************** Client Interface ***************************/

typedef void* elem;               // Elements of the Bloom filter
typedef int hash_fn(elem x);      // hash function


/****************************** Interface *******************************/

// typedef ______* bmk_t;

// Done building the Bloom filter?
bool bloommaker_frozen(bmk_t B)
  /*@requires B != NULL;                                    @*/ ;

// Create a new Bloom maker of size capacity
bmk_t bloommaker_new(int capacity)
  /*@requires 0 < capacity;                                 @*/
  /*@ensures \result != NULL && !bloommaker_frozen(\result); @*/ ;

// Freeze a Bloom maker
void bloommaker_freeze(bmk_t B)
  /*@requires B != NULL &&  !bloommaker_frozen(B);          @*/
  /*@ensures bloommaker_frozen(B);                          @*/ ;

// Adds a hash function to a Bloom maker
void bloommaker_addhash(bmk_t B, hash_fn* h)
  /*@requires B != NULL && !bloommaker_frozen(B);           @*/
  /*@requires h != NULL;                                    @*/
  /*@ensures !bloommaker_frozen(B);                         @*/ ;

// Checks an element against a finished Bloom filter
bool bloommaker_contains(bmk_t B, elem x)
  /*@requires B != NULL && bloommaker_frozen(B);            @*/ ;

// Adds an element to a finished Bloom filter
void bloommaker_add(bmk_t B, elem x)
  /*@requires B != NULL && bloommaker_frozen(B);            @*/
  /*@ensures bloommaker_contains(B, x);                     @*/ ;
```

A.4 Boolean Tables

```
/****************************** Interface *******************************/

// typedef ______* bt_t;

// Returns the capacity of a Bool table
int bt_size(bt_t T)
/*@requires T != NULL;                @*/
/*@ensures \result >= 0;              @*/ ;

// Creates a Bool table with all entries set to false
bt_t bt_new(int size)
/*@requires size >= 0;                @*/
/*@ensures \result != NULL;           @*/
/*@ensures bt_size(\result) == size;  @*/ ;

// Returns the value of the i-th entry of a Bool table
bool bt_get(bt_t T, int i)
/*@requires T != NULL;                @*/
/*@requires 0 <= i && i < bt_size(T); @*/ ;

// Sets the value of the i-th entry of a Bool table to true
void bt_set(bt_t T, int i)
/*@requires T != NULL;                @*/
/*@requires 0 <= i && i < bt_size(T); @*/ ;
```

You can also display this interface by running the terminal command

```
% cc0 -i bt.o0
```