Assignment 8
Touch Grass

Ben Grant
CSE 13S – Fall 2021

First `DESIGN.pdf` draft due: December 69th at 11:59 pm PST
Assignment due: December 420th at 11:59 pm PST
Up-to-date headers, binary, and PDF at `https://github.com/190n/asgn8`

1 Introduction

If you want to relax, watch the clouds pass by if you're laying on the grass, or sit in front of the creek; just doing nothing and having those still moments is what really rejuvenates the body.

—Miranda Kerr

Grass is one of the most prevalent types of plants on Earth. While most would picture a grassy lawn or field (see image) when they think of grass, common staples like wheat, rice, and corn are also grasses.

While grass has been known and touched throughout human history, the phrase "touch grass" has recently gained new meaning. It's used as an insult, suggesting that its target is hung up on something meaningless (especially on the Internet) and should go outside.

Those of you that have actually touched grass may know that blades of grass can be flattened with very little force, and may remain that way for some time. In this assignment, we will create data structures to represent a field of grass, and investigate several methods for flattening the entire field as quickly as possible.

A grassy field. Image by Dustin V.S. via Wikimedia Commons. CC BY-SA 3.0.

2 Is this all a joke?

Of course it is happening inside your head, Harry, but why on earth should that mean that it is not real?

—Albus Dumbledore

Of course this is a joke. Now let us continue with the assignment — grass does not touch itself!

3 Bit Vectors

Rest is not idleness, and to lie sometimes on the grass under trees on a summer's day, listening to the murmur of the water, or watching the clouds float across the sky, is by no means a waste of time.

—John Lubbock, *The Use of Life*

Flattened blades of grass. Image by S. Rae via Wikimedia Commons. CC BY 2.0.

A single blade of grass has two states: standing or flattened. This lends itself well to representation via a bit vector, as each bit also has two states.

"But Ben," you say (performance-sensitive programmer that you are), "surely this is inefficient! I have a 64-bit CPU (to say nothing of AVX and friends), so why must I touch only one blade at a time?"

In this assignment, we make an important change to the bit vector: we store the vector as an array of 64-bit integers instead of an array of bytes. This will allow us to set 64 bits at once and hopefully speed up our code. We also add a field to track how many times the bit vector has been written to, in order to quantify this speedup. Here is the `struct` definition:

```c
struct BitVector {
    uint32_t length;
    uint64_t *vector;
    uint32_t writes;
};
```

This `struct` definition *must* go in `bv.c`.

`BitVector *bv_create(uint32_t size)`

The constructor for a bit vector. `size` is the number of bits in the vector. `writes` should be initialized to zero. The vector itself should also be filled with zeros.

`void bv_delete(BitVector **bv)`

The destructor for a bit vector. All memory should be freed and the pointer set to `NULL`.

`uint32_t bv_length(BitVector *bv)`

Get the length of a bit vector.

`bool bv_set_bit(BitVector *bv, uint32_t i)`

If `i` is a valid index into the bit vector, set that bit to one and return `true`. Otherwise, return `false`. For this and the other functions that access the bit vector, you must be careful with your bitwise arithmetic since the vector stores 64-bit values. Without casting, integer literals may default to 32 bits, and this can cause operations to be performed incorrectly.

`bool bv_clr_bit(BitVector *bv, uint32_t i)`

If `i` is a valid index into the bit vector, set that bit to zero and return `true`. Otherwise, return `false`.

`bool bv_get_bit(BitVector *bv, uint32_t i, bool *bit)`

If `i` is a valid index into the bit vector, store the indicated bit in `*bit` and return `true`. Otherwise, leave the pointer unmodified and return `false`.

`bool bv_set_64(BitVector *bv, uint32_t i)`

If `i` is a valid index into the bit vector, set *every bit in the 64-bit word containing `i`* to one and return `true`. For instance, if `i` is 185, set bits 128 through 191 (index 2 in the underlying array). Since the underlying array contains 64-bit integers, this can be done with a single assignment. If `i` is not a valid index, return `false`.

`bool bv_clr_64(BitVector *bv, uint32_t i)`

This works the same as `bv_set_64`, except it sets bits to zero instead of one.

`uint32_t bv_writes(BitVector *bv)`

Get the number of writes that have been performed on a bit vector. One write is defined as one call to `bv_set_bit`, `bv_clr_bit`, `bv_set_64`, or `bv_clr_64`.

`void bv_print(BitVector *bv)`

Print the contents of a bit vector. The format is up to you.

4 Fields

His golden shield was uncovered, and lo! it shone like an image of the Sun, and the grass flamed into green about the white feet of his steed. For morning came, morning and a wind from the sea; and the darkness was removed, and the hosts of Mordor wailed, and terror took them, and they fled, and died, and the hoofs of wrath rode over them.

—J. R. R. Tolkien, *The Return of the King*

Our first data structure represents a field of grass. For simplicity, we will only consider square fields. A field, then, is an $n \times n$ matrix of blades of grass. Each blade is initially standing, and it may be flattened. Once flattened, it remains that way.

The following `struct` defines the `Field` ADT. The bit vector `matrix` is used to track the state of each blade of grass. Since a bit vector defaults to all zeros, we will use zero to represent a standing blade and one for a flattened blade.

```c
struct Field {
    uint32_t size;
    BitVector *matrix;
};
```

This `struct` definition *must* go in `field.c`.

We will implement three methods for touching grass: sequential, wide, and random. For simplicity, all three methods will have the same function signature, even though the sequential and wide methods do not need a random seed (you may use `(void) random_seed` to bypass any "unused variable" warning).

All three methods take an integer `max_iters`. They should perform no more than `max_iters` iterations (what is done on each iteration depends on the method), but they should also perform no more iterations than are necessary. For instance, on a 5×5 field, the sequential method should perform no more than 25 iterations. The random method should not account for this requirement, since doing so would require checking the whole field after every iteration.

`Field *field_create(uint32_t size)`

The constructor for a field. `size` is the length of one side of the field.

`void field_delete(Field **f)`

The destructor for a field. The field and its underlying bit vector should be freed, and the pointer should be set to `NULL`.

`uint32_t field_size(Field *f)`

Returns the field's size.

`uint32_t field_area(Field *f)`

Returns the field's area. Since we only work with square fields, this is simply the size squared.

`uint32_t field_count(Field *f)`

Returns the number of flattened blades in the field.

`uint32_t field_writes(Field *f)`

Returns the number of writes that have been performed on a field's underlying bit vector. These writes are performed by the various methods for touching grass.

`void field_touch_sequential(Field *f, uint32_t max_iters, unsigned int seed)`

Touch grass using the "sequential" method. Each iteration should mark a single blade of grass as touched using the appropriate bit vector function, starting with blade zero and continuing until `max_iters` iterations or every blade has been touched.

`void field_touch_wide(Field *f, uint32_t max_iters, unsigned int seed)`

Touch grass using the "wide" method. Each iteration should mark 64 blades of grass as touched using the appropriate bit vector, starting with blades 0-63 and continuing until `max_iters` iterations or every blade has been touched.

`void field_touch_random(Field *f, uint32_t max_iters, unsigned int seed)`

Touch grass using the "random" method. Each iteration should mark a single randomly chosen blade as touched, and you should always perform `max_iters` iterations. Touching the same blade multiple times is likely.

`void field_print(Field *f)`

Print a field of grass to standard output. Each blade is a single character: / for one that is standing, and _ for one that has been flattened, plus a single newline after each row. The bit vector should be represented in two dimensions using *row-major ordering*: blades of grass are laid out from left to right, then top to bottom. For instance, in a field of size 5, the first row is indices 0 through 4.

5 Your Task

Burn down your cities and leave our farms, and your cities will spring up again as if by magic; but destroy our farms and the grass will grow in the streets of every city in the country.

—William Jennings Bryan

Your task for this assignment is as follows:

1. Implement the two ADTs and methods for touching grass.
2. Implement a test harness for these methods. Your test harness *must* be in the file `grass.c`.
3. Test your methods, and print either the state of the field after touching grass, or various statistics about each one's performance (depending on the user's preference).

Your test harness must support any combination of the following command-line options:

- `-a`: Test all methods.
- `-s`: Test the sequential method.
- `-w`: Test the wide method.
- `-r`: Test the random method.
- `-v`: Show verbose statistics (see below).
- `-h`: Print usage information, and exit before running any tests.
- `-i iters`: Set the maximum number of iterations to run. The *default* number of iterations should be the size squared (even if the user set a custom size).
- `-n size`: Set the size of the field of grass. The *default* size should be 10. Any size from 1 to 50, inclusive, is acceptable.
- `-S seed`: Set the random seed. The *default* seed should be 7566707.

If `-v` was not passed, print each test's name followed by the contents of a field after running the test. If statistics are enabled, you should print three numbers:

- The number of blades of grass that were touched.

- The number of times that the bit vector was written to.
- The *efficiency*, a percentage representing how many blades of grass were touched by each write: $100 \times \frac{\text{touched}}{\text{writes}}$. The efficiency should be printed with two digits after the decimal point.

Here are examples of output. For further questions, consult the reference binary:

```
$ ./grass -sn 3 -i 6
Sequential:
---
---
///
$ ./grass -av
Sequential:
  Touched blades:     100
  BitVector writes:   100
  Efficiency:         100.00%
Wide:
  Touched blades:     100
  BitVector writes:     2
  Efficiency:        5000.00%
Random:
  Touched blades:      63
  BitVector writes:   100
  Efficiency:          63.00%
```

6 Deliverables

The one constant through all the years, Ray, has been baseball. America has rolled by like an army of steamrollers. It has been erased like a blackboard, rebuilt and erased again. But baseball has marked the time. This field, this game: it's a part of our past, Ray. It reminds of us of all that once was good and it could be again.

—Phil Alden Robinson, *Field of Dreams*

You will need to turn in the following source code and header files:

- `grass.c`: This contains `main()` and any other functions you write for the test harness.
- `bv.h`: Defines the interface for the bit vector ADT. Do not modify this.
- `bv.c`: Contains the implementation of the bit vector ADT.
- `field.h`: Defines the interface for the field ADT. Do not modify this.
- `field.c`: Contains the implementation of the field ADT.

You may have other source and header files, but *do not try to be overly clever*. You will also need to turn in the following:

1. `Makefile`:

    - `CC = clang` must be specified.
    - `CFLAGS = -Wall -Wextra -Werror -Wpedantic` must be specified.
    - `make` must build the `grass` executable, as should `make all` and `make grass`.
    - `make clean` must remove all files that are compiler generated.
    - `make format` should format all your source code, including the header files.

2. `README.md`: This document *must* be written using Markdown and describe how to use your program and `Makefile`, including a list of the command-line options your program accepts. You should also document any false positives reported by `scan-build` here.

3. `DESIGN.pdf`: This document *must* be a proper PDF. This design document must describe your design and design process for your program with enough detail such that a sufficiently knowledgeable programmer would be able to replicate your implementation. This does not mean copying your entire program in verbatim. You should instead describe how your program works with supporting pseudocode.

4. `WRITEUP.pdf`: This document *must* be a proper PDF. In the writeup, you should explore the characteristics of your three implementations with varying field sizes and numbers of iterations. Including graphs would be a good idea. Some specific questions that you may try to answer include:

    - Why does the "wide" method not always touch 64 blades of grass with each iteration?
    - How many iterations of the "random" method are needed to likely touch every blade of grass?
    - With the default setting of as many iterations as there are blades of grass, what portion of the field do you expect to be touched? (With some probability and calculus knowledge, you can calculate this exactly.)

7 Submission

Like previous ones, this assignment must be submitted via `git`. Send me a link to a repository that I can access (your GitLab ones are private, so either a public GitHub repository or a private one shared with `@190n` is ideal) as well as a commit ID. You may submit these through email (`bjgrant@ucsc.edu`) or Discord (`@190n#1979`).

8 Supplemental Readings

- Fast Memset and Memcpy implementations
- how to touch grass (tutorial)
- Create an empty file in Linux
- Among Drip (Music Video)
- instal mmlogi tutoril

Dfcufoaawbu wb Q wg zwys uwjwbu o acbysm o qvowbgok.