Lab 8

10 October 2023

In this lab, you'll see a bit of the experimental side of CS; in particular, measuring empirically the amount of time taken by different algorithms and data structures.

Experimenting with data structures

Today we'll see how to measure algorithm efficiency experimentally.

There are basically three pieces to this:

1. Learn how to time things,
2. set up a problem, and
3. actually time the algorithms.

This is the task for the rest of the lab period; we'll be comparing three C++ data types on the tasks of adding elements, getting elements, and removing elements. If you haven't already, start a readme file in your lab directory for this week that identifies the lab and the overall description of the task (see above); each time we add or modify an executable in the directory, add a couple lines to the readme about how to compile that executable, how to run it, and what it does. (By the end, there will be *several* executables but you should still only have one readme.)

Beginning of stuff that you read in the preview ←

Timing things

If you were measuring how long something took in real life, you'd look at the clock before it started, then again afterwards, and subtract the minutes and seconds to find the result. That's essentially what we'll do here, except that the clock we look at is the computer system's internal clock, and we'll be counting in seconds and nanoseconds.

To get started, we'll write a program that just times a simple loop. This program will have four steps: check the time once, do something, check the time again, and then compare the times.

Create the program file `timing.cpp` with the usual includes and setup, but also

```
#include <ctime>
```

Inside the `main` function, type in the following two lines:

```
timespec first_check = timespec{0, 0};
clock_gettime (CLOCK_MONOTONIC, &first_check);
```

This is a very C-ish idiom for making a system call: declare a variable (which will hold a value; in this case a clock reading), then call a function that takes the address of that variable, and afterwards the variable will have been updated with the new value. The only other relevant item here is the indication to use a monotonic clock—this is an indication to the system that we're using the clock as a stopwatch, rather than caring about the actual clock time per se.

I'll refer to that pair of lines (declaring a `timespec` variable and then calling `clock_gettime` with it) as "checking the time"; use a fresh variable each time so that you can compare them later. For now, add a second time check to the program.

Between the two time checks, add a loop that runs 1000 times. It doesn't even have to do anything; we're just killing time here.

Finally, after the second time check, you want to print the elapsed time between the checks. Before proceeding, consider for a moment how you would compute the amount of time between 3:46 and 5:10. Write out that problem, and the computation you performed to answer it, in your notebook. Take notes on the algorithm you used.

Now, I'll tell you that similar to a wall clock time (with one "field" marking hours and another marking seconds), `timespec` is a struct that contains a field `tv_sec` marking seconds and a field `tv_nsec` marking nanoseconds.[1] (There are 10^9 nanoseconds in a second.) Similar to the wall clock example, the nanoseconds "wrap around" to zero when the seconds increment; so even if you want a number of nanoseconds, you can't ignore the seconds. In your notebook, write out an expression (using `first_check.tv_sec` and

[1] So, if you had a `timespec` value called `first_check`, you would access the number of seconds as `first_check.tv_sec` . Remember, structs are basically the same thing as classes in C++.

`first_check.tv_nsec` and their counterparts from your second time check) that computes the total number of nanoseconds elapsed between the two time checks.

Type that into your program, and print out this number to `cout`. Compile and run your program.

See what happens if you vary the number of iterations in your "killing time" loop.

Modify your program to define a function `elapsed` that computes and returns the number of nanoseconds between one given `timespec` and another given `timespec`; use that function in the body of the code to print the number of nanoseconds between the two times.

Setting up a problem

New stuff starting here! ←

Ok, so now we can time things; but before we can test a data structure, we need a big bundle of data. One way to get lots of data is to generate it at random; but how to generate it? And how much to generate? In this section we'll learn how to do two useful things: get a number from the command line (to answer "how much"), and use a C++ builtin random number generator.

Start a fresh program file called `gen_data.cpp` with the usual stuff at the top, plus including `<unistd.h>`. On the first line of the `main` function, type

```
const int DATA_AMOUNT = stoi (argv[1]);
```

We've seen `argv[1]` before; it accesses the first command-line argument. The function `stoi`, which stands for "string to int", converts a string into the number it represents, e.g. `"42"` into `42`.[2]

Immediately after that, type the line

```
srandom (getpid() ^ time(nullptr));
```

This admittedly bizarre line is to ensure that the random number generator (RNG) will generate a different series of numbers every time you run the

[2] An older function, dating back to C, was called `atoi`, which you might run across from time to time. It's called the same way as `stoi` but requires a C-style string (and is short for "ASCII to int").

program.[3] Once you've run that (only once, when you first start a program), calls to a function `random()` will return a random nonnegative `long` value.

So, we need to make an array that holds `DATA_AMOUNT` items; the problem is, that number is likely to get so large that we will overflow the call stack if we declare it as

```
long data_array[DATA_AMOUNT];
```

Instead, you need to declare `data_array` dynamically, as a pointer to long, and allocate the space on the heap:

```
unique_ptr<long[]> data_array = make_unique<long[]>(DATA_AMOUNT);
```

(do you remember what you need to `#include` for `unique_ptr` to work?) but as we've discussed before, you can subsequently treat `data_array` as almost exactly equivalent to an array.

Write code that fills `data_array` with random values. Compile and run this program as well to verify that it works; obviously you can't write test cases for this (since it's *supposed* to run differently each time!), but you should print out a portion of the array to verify that it's getting populated.

Actually time the algorithms

The three tasks we'll be timing are standard ones for data structures: adding data, accessing data, and erasing data. The first data structure we'll be using is the vector; then something called a deque; then something called a map. All three are, at least the way we'll use them for now, essentially fancy upgrades of the basic array.

Begin a third program file entitled `vec_eval.cpp` as a copy of `gen_data.cpp` (since the first step will be to generate actual data).[4] You'll also need to do most of what was done in `timing.cpp`, so copy in that content as well.[5] At the top, you'll need to include `<vector>` too.

[3] Specifically, it "seeds" the RNG with a combination of the process identifier—or PID—and the number of seconds since 1970, which is what `time` returns. This should be sufficient for any casual use of an RNG.

[4] You can do this by typing `cp gen_data.cpp vec_eval.cpp`

[5] If you know about windows and buffers inside Vim, you can yank from one and paste in the other; otherwise you might try typing `:r timing.cpp` to read in the full contents of that file, and then `dd` the extraneous stuff.

After the data has been created (and stored in a plain old array), the first task is to declare a `vector<long>`. Then comes the first time check.

After the first time check comes our first testable operation. We want to run a loop that copies all the contents of `data_array` into the vector you created; the vector's method `push_back` is the usual way to do this. If your vector is named `data`, the body of your loop will thus look something like this:

```
data.push_back(data_array[i]);
```

After the data has been added to the vector, make a second time check. At this point, you should be able to compile and run this program, and it will give you an actual timing for how long it took to add all those elements to the vector. Do so, and don't forget that when you run the program you have to give it a number of data to create and use, e.g.

```
vec_eval 50000
```

(If you do that, you should get a count in the millions of nanoseconds, depending on the speed of your machine.)

Next, we want to test accessing an element in the vector. However, if we simply have a single line that accesses a single element, that instruction would run *so* fast that `clock_gettime` would have difficulty measuring it. Instead, we'll run a loop that runs 10000 times, each time through accessing the ith element.

After that loop, make a third time check. (This would be another good time to compile and make sure the code runs.)

Finally, write a loop that repeatedly (1000 times) erases the first element of the vector. This can be done with the expression

```
data.erase(data.begin());
```

depending, of course, on what you've called your vector.

After the erasures, make a fourth and final time check.

At the bottom of your program, where you do your computations of elapsed nanoseconds between the first and second, second and third, and third and fourth time checks, you might notice that the numbers they output are

large and difficult to visually process. A easier number to understand (and to compare between different runs) would be the cost of *one* call, whether adding, reading, or erasing; so when you report your elapsed nanoseconds, you should divide by the number of loop iterations it represents. (For the adding, that would be `DATA_AMOUNT` itself; for the accessing, 10000, and for the erasing, 1000.)

Once you have `vec_eval` working, try it on a few different sizes, like 100000 or 250000. (Some numbers will change more than others!)

Then, copy `vec_eval.cpp` to a file called `deque_eval.cpp`. Change occurrences of `vector` to `deque` (a different, but related, data structure); since a deque (pronounced "deck") supports all the same operations as a vector, you shouldn't have to change anything else about the program! Compile it and run it. Are the timings different?

Finally, copy `vec_eval.cpp` to a file called `map_eval.cpp`. You'll have to do slightly more work to adapt the code this time; when you declare `data` you'll need to use the type `map<int,long>` and when you add things to the map, instead of `push_back`, you'll just put the value at its associated index, e.g.

```
data[i] = data_array[i];
```

Other operations should remain the same (but their timings will, again, change).

Systematic measurement

To really see what's going on here, you need to actually take a consistent set of measurements on all three data structures. To that end, run each of the three programs on the following data sizes:

```
    10000
    50000
   100000
   500000
  1000000
  5000000
 10000000
```

(The last one is ten million, or 1 followed by seven zeroes.) Record the

results. Compare notes with other students and see if your numbers are similar (and if not, try to figure out why).

Handing in

The submission for this work will be on paper and count as a homework. Try to collect at least some of your measurements before class tomorrow (and bring them with you to class), and we'll talk about it then.

Helpful suggestions

The parameter `argc` is a count of how many command-line parameters have been given; use its value as well as the result of the `stoi` call to ensure that if the user either forgets to submit a data amount, or submits an amount < 10000, the program prints a usage message explaining the problem and exits gracefully (rather than segfaulting).

Make sure your `elapsed` function is returning a `long`, not an `int`.