ECE 2400 Computer Systems Programming, Fall 2017
Programming Assignment 2: List and Vector Data Structures

School of Electrical and Computer Engineering
Cornell University

revision: 2017-10-01-16-30

1. Introduction

The second programming assignment is designed to give you experience working with two important *data structures* in computer systems programming: lists and vectors. In this assignment, you will leverage many of the concepts from lecture including types, pointers, arrays, and dynamic allocation.

You will implement basic functions to manipulate the `list_t` and `vector_t` data structure types. These data structures both have the same high-level purpose of storing a sequence of values, but are internally organized with different approaches that heavily impact their strengths and weaknesses. Data structures are the building blocks of all software programs, so it is even more important now to design your code to be both *maintainable* and *robust*. As in the previous assignment, we will leverage the Criterion framework for unit testing, TravisCI for continuous integration testing, and Codecov.io for code coverage analysis.

After your data structures are functional and verified, you will write a 2–4 page design report that describes your design as well as each function, discusses your testing strategy, and evaluates the performance and other trade-offs between the two data structures. **Note that there is no incremental milestone for this assignment. The final code and report are all due at the end of the assignment. You should consult the programming assignment assessment rubric for more information about the expectations for all programming assignments and how they will be assessed.**

This handout assumes that you have read and understand the course tutorials. To get started, log in to an `ecelinux` machine, source the setup script, and clone your individual remote repository from GitHub:

```
% source setup-ece2400.sh
% mkdir -p ${HOME}/ece2400
% cd ${HOME}/ece2400
% git clone git@github.com:cornell-ece2400/<netid>
```

You should never fork your individual remote repository! If you need to work in isolation then use a branch within your individual remote repository. If you have already cloned your individual remote repository, then use `git pull` to ensure you have any recent updates before running all of the tests. You can run all of the tests in the lab like this:

```
% cd ${HOME}/ece2400/<netid>
% git pull --rebase
% mkdir -p pa2-dstruct/build
% cd pa2-dstruct/build
% ../configure
% make check
```

All of the tests should fail since you have not implemented the programming assignment yet. For this assignment, you will work in the `pa2-dstruct` subproject, which includes the following files:

- `configure` – Configuration script to generate Makefile
- `Makefile.in` – Makefile template to run tests and eval
- `src/list.h` – Header file for `list_t`
- `src/list.c` – Source code for `list_t`
- `src/list-test.c` – Test cases for `list_t`
- `src/list-eval.c` – Evaluation program for `list_t`
- `src/vector.h` – Header file for `vector_t`
- `src/vector.c` – Source code for `vector_t`
- `src/vector-test.c` – Test cases for `vector_t`
- `src/vector-eval.c` – Evaluation program for `vector_t`
- `src/list.dat` – Input dataset for `list_t` evaluation
- `src/vector.dat` – Input dataset for `vector_t` evaluation
- `scripts` – Scripts for the build system and generating datasets

2. First Implementation: List Data Structure

In this section, you will implement the various functions for manipulating the list data structure which is of type `list_t`. Lists are composed of nodes. Each node is of type `node_t` and contains an integer value and a pointer to the next node (see Figure 1). The pointer will be `NULL` if the node does not point to any other node. A `list_t` data structure organizes data by chaining together nodes to create a sequence of values (see Figure 2). In this assignment, our list data structure is designed to only hold a sequence of `int`s. However, we could potentially use this data structure to hold values of any other type if we changed the type of the `value` field in the definition of `node_t`. We could revise the data structure to store a sequence of `double`s or even a sequence of other lists (i.e., a list of lists)!

```
typedef struct _node_t {
  int             value;
  struct _node_t* next;
}
node_t;
```

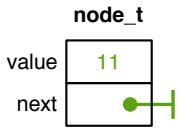

Figure 1: Definition and Example of a `node_t` Struct – The example `node_t` struct has an integer value of 11 and a next pointer which is pointing to `NULL` (i.e., does not point to any other node).

```
typedef struct {
  size_t  size;
  node_t* head;
  node_t* tail;
}
list_t;
```

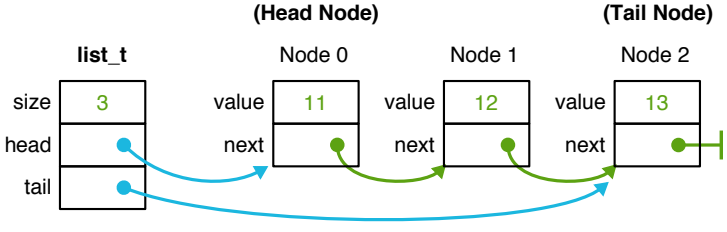

Figure 2: Definition and Example of a `list_t` Struct – The example `list_t` struct has a size of three elements, a head pointer which is pointing to Node 0, and a tail pointer which is pointing to Node 2.

Now that we know how to organize a sequence of integers as a list, we need to actually use the list. For example, we might want to add an element to the list or to search the list for a value. Although we could potentially re-write this code every time we want to use the list, it is better programming practice to refactor common code into *functions* to capture each action we might like to perform: **construct**, **destruct**, **get**, **find**, and **push back**. You are responsible for implementing each of the following functions:

```
void list_construct ( list_t* list );
void list_destruct  ( list_t* list );
int  list_get       ( list_t* list, size_t idx );
int  list_find      ( list_t* list, int value );
void list_push_back ( list_t* list, int value );
```

The specification for these functions is as follows:

- `void list_construct( list_t* list );`
  Construct the list initializing all fields in the given `list_t`. The head and tail pointers should be initialized to `NULL` to indicate that they do not point to any node.
- `void list_destruct( list_t* list );`
  Destruct the list by freeing any dynamically allocated memory used by the list and also by any of the nodes in the list.
- `int list_get( list_t* list, size_t idx );`
  Return the value at the given index (`idx`) of the list by iterating through the list using the next pointers (i.e., "following the arrows") and returning the value of the node that is at the given index. If the given index (`idx`) is out-of-bounds, the implementation should stop the program by using an assertion.
- `int list_find( list_t* list, int value );`
  Search the list for the given value (`value`) and return 1 if the value is found and 0 if it is not.
- `void list_push_back( list_t* list, int value );`
  Push a new element with the given value (`value`) at the end of the list (the tail end). You will need to dynamically allocate a new `node_t`, set its value, and correctly update the next pointer in the tail node in order to add the new node to the end of the list. You will also need to correctly update the `head` and `tail` fields in `list_t`. You can assume your implementation will never run out of memory (i.e., `malloc` will never return `NULL`).

The functions vary in complexity, and some may require just a few lines of code to implement.

Notice that each function takes as its first argument a pointer to a `list_t`. This tells the function which `list_t` to operate on. In general, you will first declare a `list_t` and then use your functions by passing in a pointer to your list. To give you an idea of how this works, here is a simple function that constructs a list, pushes back three values, gets the middle value, and then destructs the list:

```
void simple() {
  list_t mylist;                    // Declare a list_t on the stack
  list_construct( &mylist );        // Construct mylist
  list_push_back( &mylist, 11 );    // Push back 11
  list_push_back( &mylist, 12 );    // Push back 12
  list_push_back( &mylist, 13 );    // Push back 13
  int a = list_get( &mylist, 1 );   // int a now has 12
```

```
  list_destruct( &mylist );         // Destruct mylist
}
```

The definitions for `list_t` and `node_t` are provided for you in `src/list.h`. Write your implementation of each function inside `src/list.c`.

3. Second Implementation: Vector Data Structure

In this section, you will implement the various functions for manipulating the vector data structure which is of type `vector_t`. A `vector_t` data structure organizes data sequentially as a continuous chunk of memory (see Figure 3). Notice that as in a list, there is a `size` field to indicate how many elements are in the vector. However, a vector also has a `maxsize` field to indicate how big the contiguous chunk of memory is. The example vector in Figure 3 can hold five integers in a contiguous chunk of memory (i.e., `maxsize` is 5) but is only occupying the first three spaces (i.e., `size` is 3). If more than five integers need to be held, we must find a new and larger contiguous chunk of memory!

Now that we know how to organize a sequence of integers as a vector, we again want to actually use the vector. We can capture each action we want to perform into individual functions: **construct**, **destruct**, **get**, **find**, and **push back**. Notice that these provide the same functionality for vectors as our list provides. You are responsible for implementing each of the following functions:

```
void vector_construct ( vector_t* vec, size_t maxsize );
void vector_destruct  ( vector_t* vec );
int  vector_get       ( vector_t* vec, size_t idx );
int  vector_find      ( vector_t* vec, int value );
void vector_push_back ( vector_t* vec, int value );
```

The specification for these functions is as follows:

- `void vector_construct( vector_t* vec, size_t maxsize );`
  Construct the vector by initializing all fields in the `vector_t`. Notice that `vector_construct()` takes a parameter `maxsize`, which you will use to dynamically allocate space for the internal array. You can assume that `maxsize` is always "reasonable" and is never so large as to cause the machine to run out of memory (i.e., `malloc` will never return `NULL`).
- `void vector_destruct( vector_t* vec );`
  Destruct the vector by freeing any dynamically allocated memory used by the vector.

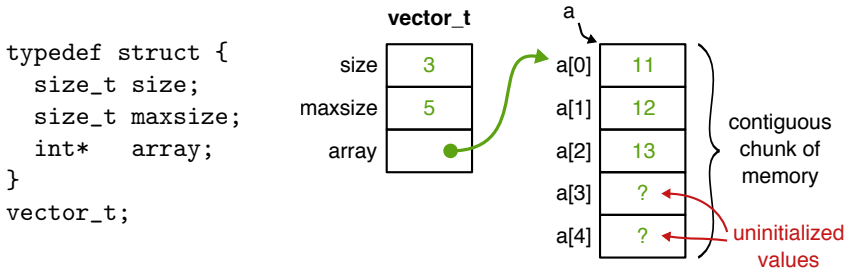

Figure 3: Definition and Example of a `vector_t` Struct – The example `vector_t` struct has a size of three elements, a maxsize of five elements, and a pointer to an internal array that holds the data.

- `int vector_get( vector_t* vec, size_t idx );`
  Return the value at the given index (`idx`) of the vector by accessing the internal array and returning the value at that index. If the given index (`idx`) is out-of-bounds, the implementation should stop the program by using an assertion.
- `int vector_find( vector_t* vec, int value );`
  Search the vector for the given value (`value`) and return 1 if the value is found and 0 if it is not.
- `void vector_push_back( vector_t* vec, int value );`
  Push a new element with the given value at the end of the vector. If there is not enough allocated contiguous space (i.e., check the `size` and `maxsize`), then you should (1) double the `maxsize`, (2) dynamically allocate space for the larger `maxsize` integers, (3) copy the data from the old space into the new space with a loop, and finally (4) free the memory in the old space. You can assume your implementation will never run out of memory (i.e., `malloc` will never return `NULL`).

The functions vary in complexity, and some may require just a few lines of code to implement.

Each function takes as its first argument a pointer to a `vector_t`. This tells the function which `vector_t` to operate on. For reference, here is a simple function that constructs a vector, pushes back three values, gets the middle value, and then destructs the vector:

```
void simple() {
  vector_t myvec;                   // Declare a vector_t on the stack
  vector_construct( &myvec, 5 );    // Construct myvec with maxsize 5
  vector_push_back( &myvec, 11 );   // Push back 11
  vector_push_back( &myvec, 12 );   // Push back 12
  vector_push_back( &myvec, 13 );   // Push back 13
  int a = vector_get( &myvec, 1 );  // int a now has 12
  vector_destruct( &myvec );        // Destruct myvec
}
```

The definition for `vector_t` is provided for you in `src/vector.h`. Write your implementation of each function inside of `src/vector.c`.

4. Testing Strategy

As in the first programming assignment, you will need to develop an effective testing strategy and write tests systematically so that you can give a compelling argument for the robustness of your code. We have provided basic tests for each of the functions you will implement. You will need to add more directed tests and random tests. Writing tests is one of the most important and challenging aspects of software programming. Designers often spend far more time designing tests than they do designing the actual program.

Design your directed tests to stress various common cases but also to capture cases that you as a programmer suspect may be challenging for your functions to handle. For example, what happens if you double the `maxsize` of a vector when `maxsize` is 0? Convince yourself that your functions for the two data structures are *robust* by carefully developing a testing *strategy*.

Random testing will be particularly useful in this programming assignment to grow your lists and vectors to arbitrary lengths, get values from random indices, and find random values that may or may not be present inside your data structure. Ensure that your random tests are repeatable by calling the `srand` function once at the top of your test case with a constant seed (e.g., `srand(0)`).

You should include assertions in your code to check for errors and to detect exceptional situations. One assertion has been provided for you that detects calling the `get` function with an index that is out-of-bounds. You can write tests that purposely fail assertions as part of your testing strategy. You can use Criterion to test for the abort signal as shown in discussion section.

5. Evaluation

Once you have verified the functionality of the list and vector implementations, you can evaluate their performance. We provide you a performance analysis harness for each implementation. You can build and run these evaluation programs like this:

```
% cd ${HOME}/ece2400/<netid>
% mkdir -p pa2-dstruct/build-eval
% cd pa2-dstruct/build-eval
% ../configure --enable-eval
% make list-eval   && ./list-eval
% make vector-eval && ./vector-eval
```

We are working in a separate `build-eval` build directory, and we are using the `--enable-eval` command line option to the `configure` script. This tells the build system to create optimized executables without any extra debugging information. You can build and run all of the evaluation programs in a single step like this:

```
% cd ${HOME}/ece2400/<netid>/pa2-dstruct/build-eval
% make eval
```

The evaluation program pushes back 2500 inputs into your data structure, starts the timer, and then uses your data structure's `find()` function to search your data structure for 5000 inputs (each returning whether or not the value is present in your data structure) before stopping the timer and reporting the total wall-clock run time. Note that of the 5000 inputs, half are present in your data structure and half are *not* present in your data structure. The inputs are not sorted in any order.

This harness will enable you to compare the performance between the list and vector data structures. Which data structure do you think will perform the `find` function more quickly? In addition to *performance*, also consider the impact of your design decisions on the *space* occupied in memory. For example, a vector that is constructed with a large initial maxsize may never need to incur the performance costs of out-growing its allocated space, but at the same time, how much memory would it occupy and potentially "waste"?

The evaluation programs also ensure that your implementations are producing the correct results, however, you should not use the evaluation programs for testing. If your implementations fail during the evaluation, then your testing strategy is insufficient. You must add more unit tests to effectively test your program before returning to performance evaluation.

Acknowledgments

This programming assignment was created by Christopher Batten and Christopher Torng as part of the ECE 2400 Computer Systems Programming course at Cornell University.