1. Introduction

The second programming assignment is designed to give you experience working with two important data structures in computer systems programming: a doubly linked list and a resizable vector. 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 two corresponding data structure types: list_int_t and vector_int_t. These data structures both have the same high-level purpose of storing a sequence of integer values, but are internally organized with different approaches that heavily impact their strengths and weaknesses. Although this programming assignment is data-structure centric, you will also need to leverage algorithms to implement the operations which manipulate the stored data. Algorithms and data structures are the building blocks of all software programs. As in the previous assignment, we will leverage the CMake framework for building programs, the CTest 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 three-page report that provides a detailed deep-dive discussion on one implementation, qualitatively evaluates trade-offs across both implementations, and quantitatively evaluates the performance across both implementations. While the final code and report are all due at the end of the assignment, we also require meeting an incremental milestone in this PA. Specific requirements for this milestone are described later in this handout. You should consult the programming assignment assessment rubric for more information about the expectations for all programming assignments and how they will be assessed. For this PA, we require you to discuss your vector implementation for the implementation deep dive in the report.

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:

```bash
% source setup-ece2400.sh
% mkdir -p ${HOME}/ece2400
% cd ${HOME}/ece2400
% git clone git@github.com:cornell-ece2400/netid
% cd ${HOME}/ece2400/netid/pa2-dstruct
% tree
```

Where netid should be replaced with your 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 working on your programming assignment.
For this assignment, you will work in the `pa2-dstruct` subproject, which includes the following files:

- `CMakeLists.txt` – CMake configuration script to generate Makefile
- `src/list.h` – Header file for `list_int_t`
- `src/list.c` – Source code for `list_int_t`
- `src/list-adhoc.c` – Ad-hoc test program for `list_int_t`
- `src/vector.h` – Header file for `vector_int_t`
- `src/vector.c` – Source code for `vector_int_t`
- `src/vector-adhoc.c` – Ad-hoc test program for `vector_int_t`
- `src/ece2400-stdlib.h` – Header file for ece2400 standard library
- `src/ece2400-stdlib.c` – Source file for ece2400 standard library
- `test/list-basic-test.c` – Basic test cases for `list_int_t`
- `test/list-directed-test.c` – Directed test cases for `list_int_t`
- `test/list-random-test.c` – Random test cases for `list_int_t`
- `test/vector-basic-test.c` – Basic test cases for `vector_int_t`
- `test/vector-directed-test.c` – Directed test cases for `vector_int_t`
- `test/vector-random-test.c` – Random test cases for `vector_int_t`
- `eval/list-push-back-eval.c` – Evaluation program for `list_int_push_back`
- `eval/list-find-eval.c` – Evaluation program for `list_int_find`
- `eval/vector-push-back-v1-eval.c` – Evaluation program for `vector_int_push_back_v1`
- `eval/vector-push-back-v2-eval.c` – Evaluation program for `vector_int_push_back_v2`
- `eval/vector-find-eval.c` – Evaluation program for `vector_int_find`
- `eval/list.dat` – Input dataset for `list_int_t` evaluation
- `eval/vector.dat` – Input dataset for `vector_int_t` evaluation

2. Implementation Specifications

You will be implementing list and vector data structures. You will need to carefully consider why you pick a specific implementation approach, and how your design and implementation choices might impact the storage requirements and performance of each data structure.

2.1. Doubly Linked List

You will implement multiple functions for manipulating a doubly linked list data structure which is of type `list_int_t`. A list is composed of nodes. Each node is of type `node_t` and contains an integer value, a pointer to the next node, and another pointer to the previous node (see Figure 1). The pointers must be `NULL` if they do not point to any other node. A `list_int_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 doubles or even a sequence of other lists (i.e., a list of lists)!
typedef struct _node_t {
    int value;
    struct _node_t* next_ptr;
    struct _node_t* prev_ptr;
} node_t;

Figure 1: Definition and Example of a node_t Struct – The example node_t struct has an integer value of 11, a next pointer, and a previous pointer. Both pointers point to NULL (i.e., do not point to any other node).

typedef struct {
    size_t size;
    node_t* head_ptr;
    node_t* tail_ptr;
} list_int_t;

Figure 2: Definition and Example of a list_int_t Struct – The example list_int_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, at, find, push back, and print. You are responsible for implementing each of the following functions:

void list_int_construct ( list_int_t* this );
void list_int_destruct ( list_int_t* this );
void list_int_push_back ( list_int_t* this, int value );
int list_int_at ( list_int_t* this, size_t idx );
int list_int_find ( list_int_t* this, int value );
void list_int_print ( list_int_t* this );

The specification for these functions is as follows:

• void list_int_construct( list_int_t* this );
  Construct an empty list and initialize all fields in the given list_int_t. The head and tail pointers should be initialized to NULL to indicate that they do not point to any node.

• void list_int_destruct( list_int_t* this );
  Destruct the list by freeing any dynamically allocated memory used by the list and also by any of the nodes in the list.

• size_t list_int_size( list_int_t* this );
  Return the current number of elements in the list. If the list is empty, this function should
return 0. It is undefined to call this function before calling list_int_construct or after calling list_int_destruct.

- void list_int_push_back( list_int_t* this, int value );
  Push a new element with the given value (value) onto the end of the list (the tail end). You should dynamically allocate one node each time list_int_push_back is called. After a new node is created, you will need to set its value, correctly update its next pointer and previous pointer, and also the tail node’s next pointer to add the new node to the end of the list. You will also need to correctly update the head_ptr and tail_ptr fields in list_int_t. You can assume your implementation will never run out of memory (i.e., ece2400_malloc will never return NULL). It is undefined to call this function before calling list_int_construct or after calling list_int_destruct.

- int list_int_at( list_int_t* this, size_t idx );
  Return the value at the given index (idx) of the list. You will need to traverse the list until you reach the given index and return the value stored in that index. Since each node has pointers to its previous and next nodes, the list can be traversed in both directions (i.e., either toward the tail node using the next pointers or toward the head node using the previous pointers). You should think about how to minimize the number of nodes you need to traverse, however, you are ultimately free to choose your own approach. If the given index (idx) is out-of-bounds, the implementation should return 0. It is undefined to call this function before calling list_int_construct or after calling list_int_destruct.

- int list_int_find( list_int_t* this, int value );
  Search the list for the given value (value) and return 1 if the value is found and 0 if it is not. If the list is empty, then the function should always return 0. You are free to choose your own approach to implement this function. Ideally, we want to minimize the number of comparisons if possible. It is undefined to call this function before calling list_int_construct or after calling list_int_destruct.

- void list_int_print( list_int_t* this );
  Print the content in the list. This function is used for for your own debugging purpose. You can implement this function in any way you like. You do not need to test this function. It is undefined to call this function before calling list_int_construct or after calling list_int_destruct.

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 this to a list_int_t. This tells the function which list_int_t to operate on. In general, you will first declare a list_int_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:

```c
void simple() {
    list_int_t lst; // Declare a list_int_t on the stack
    list_int_construct( &lst ); // Construct lst
    list_int_push_back( &lst, 11 ); // Push back 11
    list_int_push_back( &lst, 12 ); // Push back 12
    list_int_push_back( &lst, 13 ); // Push back 13
    int a = list_int_at( &lst, 1 ); // int a now has 12
    list_int_destruct( &lst ); // Destruct lst
}
```
Write the implementation of the list_int_t and node_t inside of src/list.h and the implementation of each function inside of src/list.c.

2.2. Resizable Vector

You will implement multiple functions for manipulating an vector data structure which is of type vector_int_t. The vector data structure organizes data sequentially as a contiguous chunk of memory (see Figure 3). The example vector in Figure 3 holds 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, at, find, push back, and print. Notice that these provide the same functionality for vector as our list provides. You are responsible for implementing each of the following functions:

```c
void vector_construct ( vector_int_t* this );
void vector_destruct ( vector_int_t* this );
void vector_push_back_v1 ( vector_int_t* this, int value );
void vector_push_back_v2 ( vector_int_t* this, int value );
int vector_at ( vector_int_t* this, size_t idx );
int vector_find ( vector_int_t* this, int value );
void vector_int_print ( vector_int_t* this );
```

The specification for these functions is as follows:

- **void vector_construct( vector_int_t* this );**
  Construct an empty vector by initializing all fields in vector_int_t. Both size and maxsize in vector_int_t should be initialized to 0.

- **void vector_destruct( vector_int_t* this );**
  Destruct the vector by freeing any dynamically allocated memory used by the vector.

- **size_t vector_int_size( vector_int_t* this );**
  Return the current number of elements in the vector. If the vector is empty, this function should return 0. It is undefined to call this function before calling vector_int_construct or after calling vector_int_destruct.

```c
typedef struct {
    size_t size;
    size_t maxsize;
    int* array;
} vector_int_t;
```

**Figure 3: Definition and Example of a vector_int_t Struct** – The example vector_int_t struct has a size of three elements, a maxsize of five elements, and a pointer to an internal array that holds the data.
• void vector_push_back_v1( vector_int_t* this, int value );
  Push a new element with the given value at the end of the vector. If there is not enough allocated
  contiguous space, dynamically allocate more memory to store both existing elements and the
  new element. This function allocates just enough memory (e.g., (size + 1) elements) to store both
  existing and new elements every time vector_push_back_v1 is called. You need to copy the data
  from the old space into the new space with a loop, and finally free the memory in the old space.
  In other words, size and maxsize will always be equal. You can assume your implementation
  will never run out of memory (i.e., ece2400_malloc will never return NULL). It is undefined to call
  this function before calling vector_int_construct or after calling vector_int_destruct.

• void vector_push_back_v2( vector_int_t* this, int value );
  Similar to vector_push_back_v1, this function also pushes a new element with the given value
  at the end of the vector. If there is not enough allocated contiguous space, this function doubles
  its current memory space to accommodate the new element. You also need to copy the data from
  the old space into the new space and free the old memory space. maxsize will the total amount
  of memory allocated for the vector, while size will just be the amount that is currently used. You
  can assume your implementation will never run out of memory (i.e., ece2400_malloc will never
  return NULL). It is undefined to call this function before calling vector_int_construct or after
  calling vector_int_destruct.

• int vector_at( vector_int_t* this, size_t idx );
  Return the value at the given index (idx) of the vector. If the given index (idx) is out-of-bounds,
  the implementation should return 0. It is undefined to call this function before calling vector_int_construct
  or after calling vector_int_destruct.

• int vector_find( vector_int_t* this, int value );
  Search the vector for the given value (value) and return 1 if the value is found and 0 if it is
  not. If the vector is empty, then the function should always return 0. You are free to choose
  your own approach to implement this function. As in list, we want to minimize the number of
  comparisons if possible. It is undefined to call this function before calling vector_int_construct
  or after calling vector_int_destruct.

• void vector_int_print( vector_int_t* this );
  Print the content in the vector. This function is used for for your own debugging purpose. You
  can implement this function in any way you like. You do not need to test this function. It is unde-
  fined to call this function before calling vector_int_construct or after calling vector_int_destruct.

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 this to an vector_int_t. This tells the
function which vector_int_t to operate on. For reference, here is a simple function that constructs
an vector, pushes back three values, gets the middle value, and then destructs the vector:

void simple() {
  vector_int_t vec;       // Declare a vector_int_t on the stack
  vector_construct( &vec ); // Construct an empty vec
  vector_push_back_v1( &vec, 11 ); // Push back 11
  vector_push_back_v1( &vec, 12 ); // Push back 12
  vector_push_back_v1( &vec, 13 ); // Push back 13
  int a = vector_at( &vec, 1 ); // int a now has 12
  vector_destruct( &vec );     // Destruct vec
}
The interface for the resizable vector is provided for you in src/vector.h. Write the implementation of the vector_int_t inside of src/vector.h and the implementation of each function inside of src/vector.c.

2.3. ECE 2400 Malloc and Free

Instead of using malloc function directly to allocate memory in the list and vector data structure, we provide you a pair of wrapper functions called ece2400_malloc and ece2400_free that internally call malloc and free and keep track of how much heap memory your program has allocated so far. We also provide you a utility function to return the accumulated amount of allocated heap memory. These functions are declared inside src/ece2400-stdlib.h.

• void* ece2400_malloc( size_t mem_size );
  Dynamically allocate a memory space of size mem_size in the heap. The function returns a pointer to the newly allocated space. If the allocation fails, a NULL is returned.

• void ece2400_free( void* ptr );
  Deallocate the memory space pointed by ptr in the heap. If ptr is NULL, no action occurs. Note that this function must be used in pair with ece2400_malloc, i.e., ptr must be a pointer returned by ece2400_malloc. Using this function on a pointer returned by normal malloc may result in segmentation fault.

• void ece2400_mem_reset();
  Reset the heap counter to 0.

• size_t ece2400_mem_get_usage();
  Return the accumulated amount of memory (in bytes) that has been allocated by ece2400_malloc.

3. Testing Strategy

You are responsible for developing an effective testing strategy to ensure all implementations are correct. Writing tests is one of the most important and challenging aspects of software programming. Software engineers often spend far more time implementing tests than they do implementing the actual program.

3.1. Ad-hoc Testing

To help students start testing, we provide one ad-hoc test program per implementation in src/list-adhoc.c and src/vector-adhoc.c. Students are strongly encouraged to start compiling and running these ad-hoc test programs directly in the src/ directory without using any build-automation tool (e.g., CMake and Make).

You can build and run the given ad-hoc test programs like this:

% cd ${HOME}/ece2400/netid/pa2-dstruct/src
% gcc -Wall -Wextra -pedantic-errors -o list-adhoc list.c list-adhoc.c ece2400-stdlib.c
% ./list-adhoc

% gcc -Wall -Wextra -pedantic-errors -o vector-adhoc vector.c vector-adhoc.c ece2400-stdlib.c
% ./vector-adhoc

The -Wall, -Wextra, -pedantic-errors flags will ensure that gcc reports all warnings.
3.2. Systematic Unit Testing

While ad-hoc test programs help you quickly see results of your implementations, they are often too simple to cover most scenarios. We need a systematic unit testing strategy to hopefully test all possible scenarios efficiently.

In this course, we are using CMake/CTest as a build and test automation tool. For each implementation, we provide a basic test program that checks the most basic functionality, a directed test program that should include several test cases to target different categories, and a random test program that should test that your implementation works for random inputs. **We only provide a very few directed tests and no random tests. You must add many more directed and random tests to thoroughly test your implementations!**

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. 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)`).

Before running the tests you need to create a separate `build` directory and use `cmake` to create the Makefile like this:

```bash
% cd ${HOME}/ece2400/netid/pa2-dstruct
% mkdir -p build
% cd build
% cmake ..
```

Now you can build and run all unit tests for all implementations like this:

```bash
% cd ${HOME}/ece2400/netid/pa2-dstruct/build
% make check
```

If you are failing a test program, then you can “zoom in” and run all of the unit tests for a single test program (e.g., directed tests for `list`) like this:

```bash
% cd ${HOME}/ece2400/netid/pa2-dstruct/build
% make list-directed-test
% ./list-directed-test
```

You can then “zoom in” to a specific test case by passing in the index of that test case like this:

```bash
% cd ${HOME}/ece2400/netid/pa2-dstruct/build
% make list-directed-test
% ./list-directed-test 1
% ./list-directed-test 2
```

3.3. Memory leakage

Students are also responsible for making sure that their program contains no memory leaks or other issues with dynamic allocation. We have added a new makefile target called `memcheck` which runs all of the test programs with Valgrind. Valgrind will force the test to fail if it detects any kind of memory leak or other issues with dynamic allocation.
You can check memory leaks and other issues with dynamic memory allocation for all your test programs like this:

% cd ${HOME}/ece2400/netid/pa2-dstruct/build
% make memcheck

You can just check one test program (e.g. list-basic-test) like this:

% cd ${HOME}/ece2400/netid/pa2-dstruct/build
% make list-basic-test
% valgrind --trace-children=yes --leak-check=full --error-exitcode=1 --undef-value-errors=no ./list-basic-test

Those are quite a few command line options to Valgrind, so we have created a ‘ece2400-valgrid’ script. This script is just a simple wrapper which calls Valgrind with the right options.

% cd ${HOME}/ece2400/netid/pa2-dstruct/build
% make list-basic-test
% ece2400-valgrind ./list-basic-test

### 3.4. Code Coverage

After your implementations pass all unit tests, you can evaluate how effective your test suite is by measuring its code coverage. The code coverage will tell you how much of your source code your test suite executed during your unit testing. The higher the code coverage is, the less likely some bugs have not been detected. You can run the code coverage like this:

% cd ${HOME}/ece2400/netid/pa2-dstruct
% rm -rf build-coverage
% mkdir -p build-coverage
% cd build-coverage
% cmake ..
% make check
% make coverage
% elinks coverage-html/index.html

### 4. Evaluation

Once you have verified the functionality of the list and vector implementations, you can evaluate their performance and also memory usage. We provide you with an evaluation program for the push_back and find functions: list_int_push_back, list_int_find, vector_int_push_back_v1, vector_int_push_back_v2, and vector_int_find. You can build these evaluation programs like this:

% cd ${HOME}/ece2400/netid/pa2-dstruct
% mkdir -p build-eval
% cd build-eval
% cmake -DCMAKE_BUILD_TYPE=eval ..
% make eval
Note how we are working in a separate build-eval build directory, and that we are using the
-DMAKE_BUILD_TYPE=eval command line option to the cmake script. This tells the build system
to create optimized executable without any extra debugging information. You must do your quanti-
tative evaluation using an eval build. Using a debug build for evaluation produces meaningless
results.

To run an evaluation for push back, you simply specify the number of push backs that you want to
evaluate on the command line. For example, the following runs an evaluation for 100 push backs for
the list data structure.

```bash
% cd ${HOME}/ece2400/netid
% make list-push-back-eval
% ./list-push-back-eval 100
```

To run an evaluation for find, you need to specify the number of elements that are in the list or vector.
The evaluation program will always perform 5000 finds. 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. The following runs an evaluation for 5000 finds on a 100-element list:

```bash
% cd ${HOME}/ece2400/netid
% make list-find-eval
% ./list-find-eval 100
```

The evaluation programs measure the execution time as well as the memory (heap) usage. This will
enable you to compare the performance and memory usage between list and vector. The evaluation
programs also verify that your implementations are producing the correct results. However, you
should not use the evaluation programs for testing. If your implementations fail during the evalua-
tion, then your testing strategy is insufficient. You must add more unit tests to effectively test your
program before returning to evaluation.

You should quantitatively evaluate the two find functions and three push back functions for a range
of inputs. We suggest running vector-push-back-v1-eval with input from 1 to 1000 and running
list-push-back-eval and vector-push-back-eval-v2 with input from 1 to 2500. For list-find-eval
and vector-find-eval, run them with input from 1 to 2500. Record all performance and memory
usage data and create three plots. The first plot should have the number of push backs on the x-axis
and execution time for push back on the y-axis. Plot a line for each of the three implementations
of push back. The second plot should have the number of push backs on the x-axis and the size of
heap usage on the y-axis. Plot a line for each of the three implementations of push back. The third
plot should have the number of elements of the data structure on the x-axis and execution time for
find on the y-axis. Students are required to include these three plots in their report, and to discuss
these results in the quantities evaluation section. Ensure your plots are easy to read with a legend,
reasonable font sizes, and appropriate colors/markers for black-and-white printing.

5. Incremental Milestone

While the final code and report are all due at the end of the assignment, we also require you to com-
plete an incremental milestone and push your code to GitHub on the date specified by the instructor.
In this PA, to meet the incremental milestone, you will need to write an extensive test suite including
many directed and random tests for both data structures. You do not need to complete your imple-
mentations to start writing your tests. In fact, it is a good practice to think critically about all possible
scenarios and write tests to cover them before you start working on your implementations. We will
only be checking the tests for the milestones. We will not be checking any of the implementation source code.

Acknowledgments

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