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 (dlist) and a resizable vector (rvector). 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: dlist_int_t and rvector_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. 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 CMake/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 four page report that describes each design as well as your implementation, qualitatively discusses trade-offs in each design, discusses your testing strategy, and evaluates the performance and other trade-offs between the two data structures. 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.

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
```

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 running all of the tests. You can run all of the tests in the lab like this:

```bash
% cd ${HOME}/ece2400/netid
% git pull --rebase
% mkdir -p pa2-dstruct/build
% cd pa2-dstruct/build
```
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:

- CMakeLists.txt – CMake configuration script to generate Makefile
- src/dlist.h – Header file for dlist_int_t
- src/dlist.c – Source code for dlist_int_t
- src/dlist-eval.c – Evaluation program for dlist_int_t
- src/dlist-main.c – Ad-hoc test program for dlist_int_t
- src/rvector.h – Header file for rvector_int_t
- src/rvector.c – Source code for rvector_int_t
- src/rvector-eval.c – Evaluation program for rvector_int_t
- src/rvector-main.c – Ad-hoc test program for rvector_int_t
- tests/dlist-basic-tests.c – Basic test cases for dlist_int_t
- tests/dlist-directed-tests.c – Directed test cases for dlist_int_t
- tests/dlist-random-tests.c – Random test cases for dlist_int_t
- tests/rvector-basic-tests.c – Basic test cases for rvector_int_t
- tests/rvector-directed-tests.c – Directed test cases for rvector_int_t
- tests/rvector-random-tests.c – Random test cases for rvector_int_t
- src/dlist.dat – Input dataset for dlist_int_t evaluation
- src/rvector.dat – Input dataset for rvector_int_t evaluation
- scripts – Scripts for the build system and generating datasets
- src/mem-utils.h – Header file for utility memory functions
- src/mem-utils.c – Source file for utility memory functions

2. Implementation Specifications

You will be implementing dlist and rvector 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 dlist_int_t. A dlist 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 dlist_int_t data structure organizes data by chaining together nodes to create a sequence of values (see Figure 2). In this assignment, our dlist data structure is designed to only hold a sequence of ints. 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)!

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

```c
void dlist_construct ( dlist_int_t* this );
void dlist_destruct ( dlist_int_t* this );
int dlist_at ( dlist_int_t* this, size_t idx );
int dlist_find ( dlist_int_t* this, int value );
void dlist_push_back ( dlist_int_t* this, int value );
```

The specification for these functions is as follows:

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

- **void dlist_destruct( dlist_int_t* this );**
  Destruct the dlist by freeing any dynamically allocated memory used by the dlist and also by any of the nodes in the list.

- **int dlist_at( dlist_int_t* this, size_t idx );**
  Return the value at the given index (idx) of the dlist. 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 dlist 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. You are free to choose your own approach. If the given index (idx) is out-of-bounds, the implementation should return 0.

```c
typedef struct _node_t {
    int value;
    struct _node_t* next;
    struct _node_t* prev;
} 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).

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

**Figure 2: Definition and Example of a dlist_int_t Struct** – The example dlist_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.
int dlist_find( dlist_int_t* this, int value );

Search the dlist 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.

void dlist_push_back( dlist_int_t* this, int value );

Push a new element with the given value (value) onto the end of the list (the tail end). You will need to dynamically allocate a new node_t. There are a couple of ways to do this. You can either allocate one node each time dlist_push_back is called, or allocate a group of nodes in bulk and maintain a list of "free" nodes internally. There are trade-offs in each approach. You are free to choose your own approach to implement this function. 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 and tail fields in dlist_int_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 this to a dlist_int_t. This tells the function which dlist_int_t to operate on. In general, you will first declare a dlist_int_t and then use your functions by passing in a pointer to your dlist. To give you an idea of how this works, here is a simple function that constructs a dlist, pushes back three values, gets the middle value, and then destructs the dlist:

```c
void simple() {
    dlist_int_t mylist; // Declare a dlist_int_t on the stack
    dlist_construct( &mylist ); // Construct mylist
    dlist_push_back( &mylist, 11 ); // Push back 11
    dlist_push_back( &mylist, 12 ); // Push back 12
    dlist_push_back( &mylist, 13 ); // Push back 13
    int a = dlist_at( &mylist, 1 ); // int a now has 12
    dlist_destruct( &mylist ); // Destruct mylist
}
```

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

Instead of using malloc function directly to allocate and memory, in this assignment, we provide you a wrapper function called ece2400_malloc that internally calls malloc and keeps 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. The two functions are declared inside src/mem-utils.h.

```c
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 pointer is returned.

size_t get_alloc_heap_size();
Return the accumulated amount of memory (in bytes) that has been allocated by ece2400_malloc.
```
2.2. Resizable Vector

You will implement multiple functions for manipulating the 
rvector data structure which is of type rvector_int_t. The rvector data structure organizes data sequentially as a continuous chunk of memory (see Figure 3). Notice that as in a dlist, there is a size field to indicate how many elements are in the rvector. However, an rvector 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 rvector, we again want to actually use the rvector. We can capture each action we want to perform into individual functions: construct, destruct, at, find, and push back. Notice that these provide the same functionality for rvector as our dlist provides. You are responsible for implementing each of the following functions:

```c
void rvector_construct ( rvector_int_t* this );
void rvector_destruct ( rvector_int_t* this );
int rvector_at ( rvector_int_t* this, size_t idx );
int rvector_find ( rvector_int_t* this, int value );
void rvector_push_back ( rvector_int_t* this, int value );
```

The specification for these functions is as follows:

- void rvector_construct( rvector_int_t* this );
  Construct an empty rvector by initializing all fields in rvector_int_t. Both size and maxsize in rvector_int_t should be initialized to 0.
- void rvector_destruct( rvector_int_t* this );
  Destruct the rvector by freeing any dynamically allocated memory used by the rvector.
- int rvector_at( rvector_int_t* this, 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 return 0.
- int rvector_find( rvector_int_t* this, int value );
  Search the rvector for the given value (value) and return 1 if the value is found and 0 if it is

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

**Figure 3: Definition and Example of a rvector_int_t Struct** – The example rvector_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.
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 dlist, we want to minimize the number of comparisons if possible.

- **void rvector_push_back( rvector_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 (i.e., check the size and maxsize), then you should dynamically allocate more memory to store both existing elements and the new element. Similar to dlist, there are a few ways to do this in rvector. You can allocate just enough memory (e.g., (size + 1) elements) to store both existing and new elements every time rvector_push_back is called. Or you can allocate a much larger memory space (e.g., double the maxsize). No matter what approach you choose to allocate memory for the new element, 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. 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 this to an rvector_int_t. This tells the function which rvector_int_t to operate on. For reference, here is a simple function that constructs an rvector, pushes back three values, gets the middle value, and then destructs the rvector:

```c
void simple() {
    rvector_int_t myvec; // Declare a rvector_int_t on the stack
    rvector_construct( &myvec ); // Construct an empty myvec
    rvector_push_back( &myvec, 11 ); // Push back 11
    rvector_push_back( &myvec, 12 ); // Push back 12
    rvector_push_back( &myvec, 13 ); // Push back 13
    int a = rvector_at( &myvec, 1 ); // int a now has 12
    rvector_destruct ( &myvec ); // Destruct myvec
}
```

The definition for rvector_int_t is provided for you in src/rvector.h. Write your implementation of each function inside of src/rvector.c.

As in dlist, you will use ece2400_malloc function to dynamically allocate memory in your implementation.

### 2.3. Qualitative Comparisons

Before testing and evaluating your implementations, you should qualitatively consider different approaches to implement each function for the two data structures. There is some flexibility in the way you implement the at, find, and push_back functions in both data structures. In your report you should include an explicit subsection at the end of Section 2 to discuss why you pick a specific approach to implement a particular function. What are other alternative approaches? Why do you think your approach would be better than the others? What are trade-offs in terms of time and space complexity in your approach as well as implementation complexity? Although the two data structures dlist and rvector both provide the same functionality, each uses a very different way to organize data. In this subsection, you should also discuss trade-offs of each data structure. What are advantages and disadvantages of each data structure? When should one data structure be used instead of the other?
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/dlist-main.c` and `src/rvector-main.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 program for `dlist` like this:

```
% cd ${HOME}/ece2400/<netid>/pa2-dstruct/src
% gcc -Wall -o dlist-main dlist.c dlist-main.c mem-utils.c
% ./dlist-main
```

The `-Wall` command line option 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 that checks the most basic functionality. You will need to add many 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. 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)`).

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

```
% 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:

```
% cd ${HOME}/ece2400/<netid>/pa2-dstruct/build
% make check
```
You can focus and run all of the unit tests for a single implementation (e.g., dlist) like this:

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

You can run just a single unit test for a single implementation (e.g., dlist-basic-tests) like this:

```bash
% cd ${HOME}/ece2400/<netid>/pa1-math/build
% make check-dlist-basic-tests
```

### 3.3. 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:

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

### 4. Evaluation

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

```bash
% cd ${HOME}/ece2400/<netid>
% mkdir -p pa2-dstruct/build-eval
% cd pa2-dstruct/build-eval
% cmake .. -DCMAKE_BUILD_TYPE=RELEASE
% make eval-dlist
% make eval-rvector
```

Notice that you will work in a separate build-eval directory and use -DCMAKE_BUILD_TYPE=RELEASE flag in cmake 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:

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

In each trial, the evaluation program pushes back 2500 inputs into your data structure, then uses the find function for your data structures to search for 5000 inputs (i.e., each returning whether or not the value is present in your data structure). 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.
To evaluate performance, the evaluation programs measure execution time of the first 2500 push_back function calls, the 5000 find function calls, and the overall execution of each trial. The average numbers of five trials are reported at the end of the programs.

To evaluate space efficiency, the evaluation programs measure the accumulated amount of heap memory allocated over time in the first 2500 push_back function calls. We divide those function calls into 50 bins. Each bin corresponds to 50 consecutive push_back calls. At the end of each bin, the evaluation program reads how much heap memory has been allocated so far. The evaluation program writes this data into two CSV files: dlist-eval-mem-usage.csv and rvector-eval-mem-usage.csv. Both files are generated in the build-eval directory. You can use the CSV files to plot a single memory usage plot for both dlist and rvector. The x-axis represents different bins while the y-axis shows the accumulated heap memory allocation in bytes.

Based on the performance numbers, you will should quantitatively compare and contrast the relative performance of find and push_back functions between your dlist and rvector implementations. Why is one implementation faster than the other? Does the result match your earlier qualitative analysis? If not, explain why. Based on the memory usage plot, you will discuss the space efficiency of your implementations. What data structure is more space-efficient? Is your implementation of a particular function space-efficient?

You will then relate the space efficiency to the performance of your implementations. For example, let’s say you choose to allocate memory in bulk so that your data structure incurs less performance costs when pushing back new elements. 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.

5. Incremental Milestone

While the final code and report are all due at the end of the assignment, we also require you to complete 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 implementations 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, and Tuan Ta as part of the ECE 2400 Computer Systems Programming course at Cornell University.