ECE 2400 Computer Systems Programming, Fall 2021
PA2: List and Vector Data Structures

School of Electrical and Computer Engineering
Cornell University

revision: 2021-10-04-08-38

1. Introduction

The second programming assignment will give you experience working with two important *data structures* in computer systems programming: a doubly linked list and a resizable vector. While this programming assignment is more data-structure centric, you will still need to leverage your knowledge of algorithms. Algorithms together with data structures provide the basis of all software programs. In particular, learning to analyze and compare different data structures with similar interfaces is a fundamental skill and will be tremendously useful as you continue to work with software programs.

Lists and *vectors* are data structures that are used extensively in computer systems programming, and although you will create data structures that only store integers in this programming assignment, the same approach can be used to store other types of elements as well. Although these data structures have similar interfaces, they are internally organized with different approaches that heavily impact their strengths and weaknesses. You will evaluate the impact of scaling the number of stored elements on the execution time and space usage of the various data structures. As in the previous assignment, we will leverage the CMake framework for building programs, the CTest framework for unit testing, and GitHub Actions for continuous integration testing.

After your data structures are functional and tested, you will write a four-page report that includes your complexity analysis and a quantitative evaluation of the performance across all implementations. **You should consult the programming assignment logistics document for more information about the expectations for all programming assignments and how they will be assessed. 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. Requirements specific to this PA for the incremental milestone and the final submission are described at the end of this handout.**

This handout assumes that you have read and understand the course tutorials and that you have attended the discussion sections. To get started, log in to an `ecelinux` server, 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
% cd ${HOME}/ece2400/netid/pa2-dstruct
% tree
```

Where `netid` should be replaced with your NetID. You can both pull and push to your individual remote repository. **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.

```
% cd ${HOME}/ece2400/netid
% git pull
% tree pa2-dstruct
```

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/ece2400-stdlib.h`	– Header file for course standard library
• `src/ece2400-stdlib.c`	– Source code for course standard library
• `src/list-int.h`	– Header file for `list_int_t`
• `src/list-int.c`	– Source code for `list_int_t`
• `src/list-int-adhoc.c`	– Ad-hoc test program for `list_int_t`
• `src/vector-int.h`	– Header file for `vector_int_t`
• `src/vector-int.c`	– Source code for `vector_int_t`
• `src/vector-int-adhoc.c`	– Ad-hoc test program for `vector_int_t`
• `test/list-int-directed-test.c`	– Directed test cases for `list_int_t`
• `test/list-int-random-test.c`	– Random test cases for `list_int_t`
• `test/vector-int-directed-test.c`	– Directed test cases for `vector_int_t`
• `test/vector-int-random-test.c`	– Random test cases for `vector_int_t`
• `eval/list-int.dat`	– Input dataset for `list_int_t` evaluation
• `eval/vector-int.dat`	– Input dataset for `vector_int_t` evaluation
• `eval/list-int-push-back-eval.c`	– Evaluation program for `list_int_push_back`
• `eval/list-int-contains-eval.c`	– Evaluation program for `list_int_contains`
• `eval/vector-int-push-back-v1-eval.c`	– Evaluation program for `vector_int_push_back_v1`
• `eval/vector-int-push-back-v2-eval.c`	– Evaluation program for `vector_int_push_back_v2`
• `eval/vector-int-contains-eval.c`	– Evaluation program for `vector_int_contains`

The programming assignment is divided into the following steps. Complete each step before moving on to the next step.

- Step 1. Implement and test `list_int_construct` and `list_int_destruct`
- Step 2. Implement and test `list_int_push_back`, `list_int_size`, and `list_int_at`
- Step 3. Implement and test `list_int_contains`
- Step 4. Implement and test `vector_int_construct` and `vector_int_destruct`
- Step 5. Implement and test `vector_int_push_back`, `vector_int_size`, and `vector_int_at`
- Step 6. Implement and test `vector_int_contains`
- Step 7. Evaluate all implementations

We cannot stress enough how important it is to take an incremental design approach! You really must implement *and test* each function before trying to move on to the next function. This means more than just adhoc testing. You must do thorough directed testing of each function *before* implementing the next function. Do not just implement all of the functions and then start testing.

2. Interface and Implementation Specifications

You will be implementing list and vector data structures to store integer values. 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.

Note that your implementations cannot use anything from the Standard C library except for the `printf` function defined in `stdio.h`, the MIN/MAX macros defined in `limits.h`, the `NULL` macro defined in `stddef.h`, and the `assert` macro defined in `assert.h`. You should not use `malloc` and `free` functions directly, but should instead be using `ece2400_malloc` and `ece2400_free`.

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 `list_int_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 `list_int_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)!

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 the following *functions* to capture each action we might like

```
typedef struct _list_int_node_t {
  int                      value;
  struct _list_int_node_t* next_p;
  struct _list_int_node_t* prev_p;
}
list_int_node_t;
```

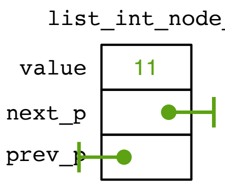

Figure 1: Definition and Example of a `list_int_node_t` Struct – The example `list_int_node_t` struct has an integer value of 11, a next pointer, and a previous pointer. Both pointers point to `NULL` (i.e., do not point to any other node).

```
typedef struct {
  list_int_node_t* head_p;
  list_int_node_t* tail_p;
  int              size;
}
list_int_t;
```

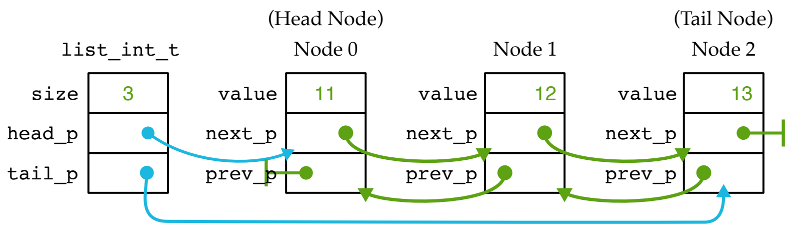

Figure 2: Definition and Example of a `list_int_t` Struct – The example `list_int_t` struct has a size of three elements, a head pointer which is pointing to Node 0, and a tail pointer which is pointing to Node 2.

to perform: **construct**, **destruct**, **push back**, **size**, **at**, **contains**, 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_size      ( list_int_t* this  );
int  list_int_at        ( list_int_t* this, int idx   );
int  list_int_contains  ( 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. It is undefined to call this function more than once on the same list.
- `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. It is undefined to call this function more than once on the same list.
- `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_p` and `tail_p` 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 `construct` or after `destruct`.
- `int 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 `construct` or after `destruct`.
- `int list_int_at( list_int_t* this, int 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. If the given index (`idx`) is out-of-bounds, the implementation should return 0. It is undefined to call this function before `construct` or after `destruct`.
- `int list_int_contains( 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. It is undefined to call this function before `construct` or after `destruct`.

- `void list_int_print( list_int_t* this );`
  Print the contents of the list. This function is used 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 `construct` or after `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. The behavior of all the functions above is undefined if the `this` pointer is `NULL` or points to an invalid `list_int_t` struct.

To give you an idea of how this works, here is a simple program that constructs a list, pushes back three values, gets the middle value, and then destructs the list:

```
int main( void )
{
  list_int_t lst;                  // Declare a list_int_t on the stack
  list_int_construct ( &lst );     // Construct an empty list
  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
  return 0;
}
```

The interface for the doubly linked list is provided for you in `src/list-int.h`. Write the implementation of `list_int_t` and `list_int_node_t` in `src/list-int.h` and the implementation of each function inside of `src/list-int.c`.

2.2. Resizable Vector

You will implement multiple functions for manipulating a vector data structure which is of type `vector_int_t`. The vector data structure organizes data sequentially as a continuous 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**, **push back**, **size**, **at**, **contains**, 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:

```
void vector_int_construct    ( vector_int_t* this  );
void vector_int_destruct     ( vector_int_t* this  );
void vector_int_push_back_v1 ( vector_int_t* this, int value );
void vector_int_push_back_v2 ( vector_int_t* this, int value );
int  vector_int_size         ( vector_int_t* this  );
int  vector_int_at           ( vector_int_t* this, int idx   );
int  vector_int_contains     ( vector_int_t* this, int value );
void vector_int_print        ( vector_int_t* this  );
```

```
typedef struct {
  int* data;
  int  maxsize;
  int  size;
}
vector_int_t;
```

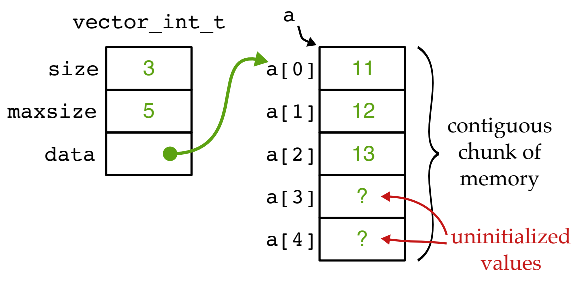

Figure 3: Definition and Example of a `vector_int_t` Struct – The example `vector_int_t` struct has a size of three elements, a maxsize of five elements, and a pointer to an internal array that holds the data.

The specification for these functions is as follows:

- `void vector_int_construct( vector_int_t* this );`
  Construct an empty vector by initializing all fields in `vector_int_t`. `size` should be initialized to 0. `maxsize` should be initialized appropriately given the rest of the implementation. It is undefined to call this function more than once on the same vector.

- `void vector_int_destruct( vector_int_t* this );`
  Destruct the vector by freeing any dynamically allocated memory used by the vector. It is undefined to call this function more than once on the same vector.

- `int 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 `construct` or after `destruct`.

- `void vector_int_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 should allocate just enough memory (e.g., (size + 1) elements) to store both existing and new elements. 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., `ece2400_malloc` will never return `NULL`). It is undefined to call this function before `construct` or after `destruct`.

- `void vector_int_push_back_v2( vector_int_t* this, int value );`
  Similar to `vector_int_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 just be 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 `construct` or after `destruct`.

- `int vector_int_at( vector_int_t* this, int 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 `construct` or after `destruct`.

- `int vector_int_contains( 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. We want to minimize the number of comparisons if possible. It is undefined to call this function before `construct` or after `destruct`.
- `void vector_int_print( vector_int_t* this );`
  Print the content in the vector. This function is used 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 `construct` or after `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`. In general, you will first declare a `vector_int_t` and then use your functions by passing in a pointer to your vector. This tells the function which `vector_int_t` to operate on. The behavior of all the functions above is undefined if the `this` pointer is `NULL` or points to an invalid `vector_int_t` struct.

For reference, here is a simple program that constructs a vector, pushes back three values, gets the middle value, and then destructs the vector:

```
int main( void )
{
  vector_int_t vec;                    // Declare a vector_int_t on the stack
  vector_int_construct   ( &vec );     // Construct an empty vector
  vector_int_push_back_v1( &vec, 11 ); // Push back 11
  vector_int_push_back_v1( &vec, 12 ); // Push back 12
  vector_int_push_back_v1( &vec, 13 ); // Push back 13
  int a = vector_int_at  ( &vec, 1 );  // int a now has 12
  vector_int_destruct    ( &vec );     // Destruct vec
  return 0;
}
```

The interface for the resizable vector is provided for you in `src/vector-int.h`. Write the implementation of `vector_int_t` in `src/vector-int.h` and the implementation of each function in `src/vector-int.c`.

2.3. ECE 2400 Malloc and Free

Instead of using the `malloc` function directly for dynamic memory allocation in the list and vector data structures, we provide you a pair of wrapper functions called `ece2400_malloc` and `ece2400_free`. These functions are declared inside `src/ece2400-stdlib.h`. These two functions internally call `malloc` and `free`, but they also keep track of how much heap memory your program has allocated so far.

- `void* ece2400_malloc( size_t mem_size );`
  Dynamically allocate a memory space of size `mem_size` on the heap. The function returns a pointer to the newly allocated space. If the allocation fails, a `NULL` is returned. Note that just like `malloc`, this function has a parameter of type `size_t`. Because we use the `-Wconversion` flag to tell the compiler to warn of us of any potentially unsafe implicit type conversions, this means we need to explicitly cast any variables of time `int` to `size_t` when calling this function. See an example below.
- `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` is undefined and may result in a segmentation fault

For reference, here is a simple function that allocates an array of N integers on the heap.

```
int main( void )
{
  int  N    = 32;
  int* data = ece2400_malloc( (size_t) N * sizeof(int) );

  // ... do something with data ...

  ece2400_free( data );
  return 0;
}
```

Notice the need to use an explicitly cast the variable N to `size_t`. Technically this means if N is negative you will allocate a huge amount of memory on the heap, so you should ensure that N is not negative.

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.

Note that while there are limitations on what you can use from the Standard C library in your *implementations* there are no limitations on what you can use from the Standard C library in your *testing strategy*. You should feel free to use the Standard C library in your golden reference models and/or for random testing.

3.1. Ad-hoc Testing

To help students start testing, we provide one ad-hoc test program per implementation in `src/list-int-adhoc.c` and `src/vector-int-adhoc.c`. Students are 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 -o list-int-adhoc ece2400-stdlib.c list-int.c list-int-adhoc.c
% ./list-int-adhoc

% gcc -Wall -o vector-int-adhoc ece2400-stdlib.c vector-int.c vector-int-adhoc.c
% ./vector-int-adhoc
```

The `-Wall` flag will ensure that `gcc` reports most 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 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 in 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)`).

We provide you a testing framework you should use for your directed and random testing. See the provided test programs in the `test` subdirectory for how to use this framework. The ECE 2400 standard library in `ece2400-stdlib.h` contains the following macros you should use to check the correctness of your implementations:

- `ECE2400_CHECK_FAIL()` – check program does not reach this point
- `ECE2400_CHECK_TRUE( expr_ )` – check `expr_` is always true
- `ECE2400_CHECK_FALSE( expr_ )` – check `expr_` is always false
- `ECE2400_CHECK_INT_EQ( expr0_, expr1_ )` – check `expr0_` equals `expr1_`

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
```

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:

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

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

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

3.3. Test-Case Crowd Sourcing

While a comprehensive test suite provides strong evidence that your implementation has the correct functionality, it is particularly challenging to write high-quality test cases for all of your implementations. **Students can use test-case crowd-sourcing after the milestone to reduce the workload of constructing a comprehensive test suite.** Test-case crowd-sourcing will use a Canvas discussion page; students cannot see any of the currently posted test cases until they post one of their own. Focus on uploading one or two very strong directed or random test cases. Do not upload more than two test cases. Avoid uploading simple directed test cases since students will have already developed such test cases for the milestone. Posting the basic test case provided by the course instructors, posting an obviously too simple test case, and/or posting something which is obviously meant to "game" the system is not allowed. Let's all work together to crowd-source a great test suite that every student can take advantage of!

You can use test cases posted in the Canvas discussion page in your test programs as long as you acknowledge the author, so be sure to include the comment in your source code which describes the test case and includes the author's name. You will need to renumber the test cases and call them correctly from `main()`. **Make sure you understand the test case and that you feel it is testing correct behavior before including it in your test suite!**

3.4. Memory Leaks

Students are also responsible for making sure that their program contains no memory leaks or other issues with dynamic allocation. We have included a make 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-int-directed-test`) like this:

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

Those are quite a few command line options to Valgrind, so we have created an `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-int-directed-test
% ece2400-valgrind ./list-int-directed-test
```

3.5. 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
```

Note that these code coverage results will reflect *all* prior runs of the test and evaluation programs in the build directory. That is why in the above example, we do a fresh build in a separate `build-coverage` build directory.

If you want to drill down and explore the coverage of each line in a program you use use the elinks web browser like this:

```
% cd ${HOME}/ece2400/netid/pa2-dstruct/build-coverage
% elinks coverage-html/index.html
```

Code coverage is just one more piece of evidence you can use to make a compelling case for the correct functionality of your implementations. It is not required that students achieve 100% code coverage. It is far more important that students simply use code coverage as a way to guide their test-driven design than to overly focus on the specific code coverage number.

4. Evaluation

Once you have tested 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 `contains` functions: `list_int_push_back`, `list_int_contains`, `vector_int_push_back_v1`, `vector_int_push_back_v2`, and `vector_int_contains`. You should not need to modify the aevaluation programs. The ECE 2400 standard library in `ece2400-stdlib.h` contains the following functions that are used in the evaluation programs to measure the execution time and heap space usage.

• `ece2400_timer_reset()`	– reset global timer
• `ece2400_timer_get_elapsed()`	– return elapsed time in seconds since last reset
• `ece2400_mem_reset()`	– reset global memory usage counter
• `ece2400_mem_get_aux_usage()`	– return max heap space allocated in bytes since last reset

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 `-DCMAKE_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 quantitative 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.

```
% cd ${HOME}/ece2400/netid/pa2-dstruct/build-eval
% make list-int-push-back-eval
% ./list-int-push-back-eval 100
```

To run an evaluation for contains, you need to specify the number of elements that are in the list or vector. The evaluation program will always perform 5000 calls to the `contains` function, with the argument to `contains` uniformly randomly chosen from the values present in the data structure. The inputs are not sorted in any order. The following runs an evaluation for 5000 contains on a 100-element list:

```
% cd ${HOME}/ece2400/netid/pa2-dstruct/build-eval
% make list-int-contains-eval
% ./list-int-contains-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 evaluation, 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 three push back functions and two contains functions for a range of inputs. We suggest running the `list-int-push-back-eval`, `vector-int-push-back-v1-eval`, and `vector-int-push-back-v2-eval` with input from 100 to 2000. For `list-int-contains-eval` and `vector-int-contains-eval`, run them with input from 100 to 2000. Record all of this performance data.

5. Milestone and Final Submission

This section includes critical information about the incremental milestone, final code submission, and the final report specific to this PA. **The programming assignment logistics document provides general details about the requirements for the milestone and final submission.** You must actually read the document to ensure you know how we will access your milestone and final submission.

5.1. 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 by the date specified by the instructor. In this PA, to meet the incremental milestone, you will need to implement the list and write an extensive test suite including many directed and random tests for this implementation. Here is how we will be testing your milestone:

```
% mkdir -p ${HOME}/ece2400/submissions
% cd ${HOME}/ece2400/submissions
% rm -rf repo
% git clone git@github.com:cornell-ece2400/netid

% cd ${HOME}/ece2400/submissions/netid/pa2-dstruct
% mkdir -p build
% cd build
% cmake ..
% make check-milestone
```

5.2. Final Code Submission

Your code quality score will be based on the way you format the text in your source files, proper use of comments, deletion of instructor comments, and uploading the correct files to GitHub (only source files should be uploaded, no generated build files). To assist you in formatting your code correctly, we have created a make target that will autoformat the code for you. You can use it like this:

```
% cd ${HOME}/ece2400/netid/pa2-dstruct
% mkdir -p build
% cd build
% cmake ..
% make autoformat
% git diff
# ... check all changes ...
% git commit -a -m "autoformat"
```

Note that the `autoformat` target will only work if you have already committed all of your work. This way you can easily use `git diff` to view the changes made by the autoformatting and commit those changes when you are happy with them. Since we provide students an automated way to format their code correctly, students have no excuse for not following the course coding conventions!

Note that students must remove unnecessary comments that are provided by instructors in the code distributed to students. Students must not commit executable binaries or any other unnecessary files. The `autoformat` target will not take care of these issues for you.

To submit your code you simply upload it to GitHub. Your code will be assessed both in terms of functionality and code quality. Your functionality score will be determined by running your code against a series of tests developed by the instructors to test its correctness. Here is how we will be testing your final code submission:

```
% mkdir -p ${HOME}/ece2400/submissions
% cd ${HOME}/ece2400/submissions
% rm -rf repo
% git clone git@github.com:cornell-ece2400/netid

% cd ${HOME}/ece2400/submissions/netid/pa2-dstruct
% mkdir -p build
% cd build
% cmake ..
% make check
% make eval
# ... run the eval programs ...
```

5.3. Final Report

The final report must be uploaded to Canvas. The date you upload your report will indicate how many slip days you are using for the assignment. For this PA, we require you to include four sections: introduction, complexity analysis, quantitative evaluation, and conclusion.

The complexity analysis section of your report must include a table that summarizes the time and space complexity (in big-O notation) of several functions (see Table 1 for a template). For time complexity analysis, you need to pick a key operator. For space complexity analysis, you need to analyze the *auxillary heap space usage* of *just* that function (i.e., do not include the heap space usage of the data-structure before calling the function). The input parameter is N where N is the number of elements stored in the data structure. This means your complexity analysis should capture the trend as we call the function on larger and larger data-structures. *Best/worst case complexity analysis* for the `at` function should consider the best/worst case values of the given index (`idx`). *Average case complexity analysis* for `contains` should assume the function is called with a value chosen from the values present in the data structure using a uniform random distribution. *Amortized complexity analysis* for `push_back` should assume a scenario where you want to fill an empty data structure with N elements by calling `push_back` N times. Then analyze the *amortized* cost of each `push_back` call as discussed in lecture. Justify your entries in the table in the complexity analysis section. Note that you don't need to explicitly discuss all six steps of complexity analysis and we are not looking for a rigorously formal proof, but you do need to be clear about the assumptions you made during analysis and provide some kind of compelling high-level argument.

The quantitative evaluation section of your report must include three plots of execution time and auxillary heap space usage. You should create the plots using the data recorded from your quantitative evaluation. The first plot should have the number of push backs on the x-axis and *amortized execution time in microseconds* for doing that number of push backs on the y-axis. Plot a data series for each of the three implementations (`list-int`, `vector-int-v1`, `vector-int-v2`) of push back. The second plot should have the number of push backs on the x-axis and the *amortized auxillary heap space usage* in bytes on the y-axis. Plot a data series for each of the three implementations of push back. The third plot should have the number of elements stored in the data structure on the x-axis and average execution time in microseconds per call to `list_int_contains` and `vector_int_contains` on the y-axis. Ensure your plots are easy to read with a legend, reasonable font sizes, and appropriate colors/markers for black-and-white printing. The quantitative evaluation section of your report must describe how you collected this data and what conclusions can be drawn from this data.

The quantitative evaluation section of your report must also include a table reporting a best-fit linear or polynomial equation as a function of N for each data series determined using a tool of your choice (see Table 2 for a template). The equation should be in units of microseconds for execution time and in units of bytes for auxillary heap space usage. Any term that is less than 0.001 µs should be rounded to zero. For example, if the best fit for the execution time data is a quadratic equation of the form $10.48N^2 + 7.32\text{e-}04N + 31.6$ in microseconds, then the appropriate equation for the table would be $10.48N^2 + 31.6$ (i.e., the constant factor for the N term is effectively zero). **The quantitative evaluation section of your report must discuss the connection between your theoretical complexity analysis and your experimental data as captured by these best-fit linear or polynomial regressions.**

		Time Complexity		**Space Complexity**	
Function	**Analysis**	`list_int`	`vector_int`	`list_int`	`vector_int`
`push_back_v1`	amortized				
`push_back_v2`	amortized	n/a		n/a	
`size`	worst				
`at`	best				
`at`	worst				
`contains`	average				

Table 1: Template for Complexity Analysis Table

		Execution Time (µs)		**Aux Heap Space Usage (B)**	
Function	**Analysis**	`list_int`	`vector_int`	`list_int`	`vector_int`
`push_back_v1`	amortized				
`push_back_v2`	amortized	n/a		n/a	
`contains`	average				

Table 2: Template for Measured Execution Time and Space Usage Equation Table

Acknowledgments

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