ECE 2400 Computer Systems Programming, Spring 2025
PA2: List and Vector Data Structures

School of Electrical and Computer Engineering
Cornell University

revision: 2025-02-23-14-35

Clarifications or fixes to this writeup after the initial publication will appear in this color.

1. Introduction

This programming assignment will give you experience working with two important data structures in computer systems programming: a **doubly linked list** and a **resizable vector**. You will leverage your knowledge of basic algorithms. You will learn to analyze and compare different data structures with similar interfaces. Skills like these will set you apart from programmers who write "code that works" and bring you into a league of sophisticated programmers who craft elegant, efficient code.

Lists and *vectors* are two data structures that are used extensively in computer systems programming. Although you will handle only integer data in this assignment, the lessons learned apply to other types of elements. By the end of this assignment, you will understand the implementation details that show the contrasting strengths and weaknesses of *lists* versus *vectors*. You will evaluate the impact of scaling the number of elements on the execution time and memory usage.

After your data structures are functional and tested for memory leaks, you will write a four-page report that includes the complexity analysis and a quantitative evaluation of performance across all 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. Requirements specific to this PA's milestone and final submission are described at the end of this handout.**

To get started, log into an `ecelinux` server and then use `git pull` to ensure you have any recent updates (i.e., the `pa2-dstruct` release code) 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
|-- eval                                    programs for evaluating your implementations
|   |-- CMakeLists.txt                          cmake configuration script
|   |-- list-int-contains-eval.c                evaluates list_int_contains
|   |-- list-int-push-back-eval.c               evaluates list_int_push_back
|   |-- vector-int-contains-eval.c              evaluates vector_int_contains
|   |-- vector-int-push-back-v1-eval.c          evaluates vector_int_push_back_v1
|   |-- vector-int-push-back-v2-eval.c          evaluates vector_int_push_back_v2
```

```
|-- include                             header and data files
|   |-- ece2400-stdlib.h                  header file for course standard library
|   |-- list-int.dat                      input dataset for list_int_t evaluation
|   |-- list-int.h                        header file for list_int_t
|   |-- vector-int.dat                    input dataset for vector_int_t evaluation
|   |-- vector-int.h                      header file for vector_int_t
|-- README.md                           HOW TO GET STARTED!
|-- scripts                             scripts to accomplish all sorts of tasks
|   |-- build.sh                          compiles your code
|   |-- coverage.sh                       assesses the coverage of your tests
|   |-- eval.sh                           builds, tests, then evaluates
|   |-- format.sh                         formats code according to class conventions
|   |-- memcheck.sh                       looks for memory leaks!
|   |-- test.sh                           compiles and then tests for correctness
|   |-- valgrind.sh                       user-friendly way to run Valgrind
|-- src                                 source code – where the magic happens
|   |-- CMakeLists.txt                    cmake configuration script
|   |-- ece2400-stdlib.c                  source code for course standard library
|   |-- list-int-adhoc.c                  ad-hoc test program for list_int_t
|   |-- list-int.c                        source code for list_int_t
|   |-- vector-int-adhoc.c                ad-hoc test program for vector_int_t
|   |-- vector-int.c                      source code for vector_int_t
|-- test                                correctness tests
    |-- CMakeLists.txt                    cmake configuration script
    |-- list-int-directed-test.c          directed test cases for list_int_t
    |-- list-int-random-test.c            random test cases for list_int_t
    |-- vector-int-directed-test.c        directed test cases for vector_int_t
    |-- vector-int-random-test.c          random test cases for vector_int_t
```

This assignment is divided into seven 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

Take an incremental design approach! Implement *and test* each function before trying to move on to the next function. This means more than just adhoc testing. You must perform *thorough* directed testing of each function *before* implementing the next function. If you implement all of the functions and **then** start testing, you will experience a Sarlacc Pit-level of pain and suffering.

2. Interface and Implementation Specifications

You will implement list and vector data structures to store integer values. You will need to carefully consider each specific implementation approach, and how your design choices might impact storage requirements and performance.

Your implementations cannot use anything from the Standard C library except for (1) the `printf` function defined in `stdio.h`, (2) the MIN/MAX macros defined in `limits.h`, (3) the `NULL` macro defined in `stddef.h`, and (4) the `assert` macro defined in `assert.h`. Do **not** use `malloc` and `free` functions directly. Instead, use `ece2400_malloc` and `ece2400_free`.

2.1. Doubly Linked List

You will implement multiple functions to manipulate a doubly linked list data structure of type `list_int_t`. A list is comprised 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.

```c
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;
```

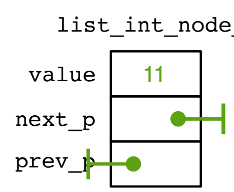

Figure 1: Definition and Example of a `list_int_node_t` struct. The example has a value of `11`. The next and previous pointers both point to `NULL` (i.e., do not point to any other node).

A `list_int_t` data structure organizes data by chaining nodes together to create a sequence of values (see Figure 2). In this assignment, our list data structure is designed to hold only 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)!

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

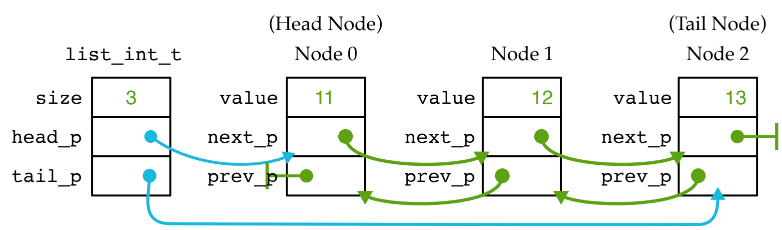

Figure 2: Definition and Example of a `list_int_t` struct. The example has a size of three elements, a head pointer which is pointing to Node 0, and a tail pointer which is pointing to Node 2.

Now that we know how to organize a sequence of integers as a list, we need to use the list. For example, we might want to add an element to the list or 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 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 tail end of the list. Dynamically allocate one node each time `list_int_push_back` is called. After a new node is created, 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. 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. Traverse the list until you reach the given index and return the value stored in that index. Since each node has pointers to its prev 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 prev pointers). 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 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 debugging purposes. 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 linked list is provided in `include/list-int.h`. Write the struct definitions in `include/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 occupying only 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!

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

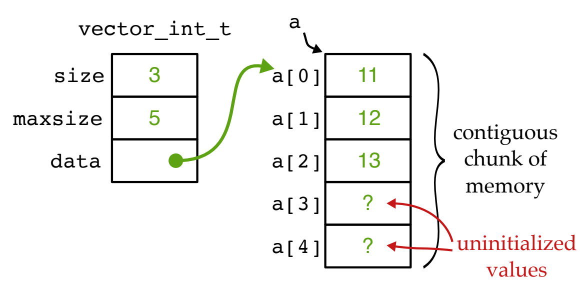

Figure 3: Definition and Example of a `vector_int_t` struct. The example has a size of three elements, a maxsize of five elements, and a pointer to an internal array that holds the data.

Once again, we can capture each action we want to perform into individual functions: **construct**, **destruct**, **push back**, **size**, **at**, **contains**, and **print**. 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  );
```

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. Allocate just enough memory (e.g., (size + 1) elements) to store both existing and new elements. 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. Copy the data from the old space into the new space and free the old memory space. `maxsize` will 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, 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. 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 debugging purposes. 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. Some may require just a few lines of code to implement. 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.

This program constructs a vector, pushes 3 values, gets the middle value, then destructs the vector:

```
int main( void )
{
  vector_int_t vec;                // Declare a vector_int_t on 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 `malloc` and `free`, **you should use the wrapper functions** `ece2400_malloc` and `ece2400_free`, declared in `include/ece2400-stdlib.h` and implemented in `src/ece2400-stdlib.c`. They internally call `malloc` and `free` and track how much heap memory your program has allocated.

- `void* ece2400_malloc( size_t mem_size );`

  Dynamically allocates 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. 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, we need to explicitly cast any variables of type `int` to `size_t` when calling this function. See example below.

- `void ece2400_free( void* ptr );`

  Deallocates the memory space pointed by `ptr` in the heap. If `ptr` is `NULL`, no action occurs. 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` were negative, `ece2400_malloc` will allocate a *huge* amount of memory on the heap! So you should ensure that `N` is not negative.

3. Testing Strategy

Develop 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 more time implementing tests than they do implementing the actual program.

Although 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*. Feel free to use the Standard C library in your golden reference models and/or for random testing.

3.1. Ad-hoc (aka Smoke) Testing

To help students test as they code, 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 running these ad-hoc test programs directly like this:

```
% cd ${HOME}/ece2400/netid/pa2-dstruct
% scripts/build.sh
% cd build/src
% ./list-int-adhoc
```

3.2. Systematic Unit Testing

Although ad-hoc test programs help you quickly see results of your implementations, they will not robustly cover most scenarios. We need a systematic unit testing strategy to hopefully test all possible scenarios efficiently.

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 provide only 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)`).

As in the previous programming assignment, 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_`

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

```
% cd ${HOME}/ece2400/netid/pa2-dstruct
% scripts/test.sh
```

```
Build Successful
<blah blah blah>
Tests failed
```

(Your tests will all fail initially.)

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/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/test
% ./list-int-directed-test 2
test_case_2_simple_contains
<blah blah blah>
```

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. After the milestone deadline, **students can use test-case crowd-sourcing 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 be honest, the test cases many of you uploaded for PA1 were pretty sad. If the tests you upload for PA2 are similarly lackluster, we won't continue doing this because we don't see the point.*

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 script 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
% ./scripts/memcheck.sh
```

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

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

Our `valgrind.sh` script calls Valgrind with the correct command line options so you don't need to remember all of them.

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. **However, achieving full code coverage does not guarantee that your tests are correct or passing.** To generate coverage reports, run the following from `pa2-dstruct`:

```
% ./scripts/coverage.sh
```

The script will clean up any previous coverage data, create a fresh `build-coverage` directory, compile the project with code coverage flags, run the tests, and generate coverage reports. The coverage reports for your list and vector implementations can be found at `build-coverage/list-int.c.gcov` and `build-coverage/vector-int.c.gcov`. Unexecuted lines are marked #####. Lines marked with * contain some unexecuted basic blocks.

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 evaluation 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
% ./scripts/eval.sh
```

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
% scripts/build.sh
% build/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:

```
% scripts/build.sh
% build/eval/list-int-contains-eval 100
```

The evaluation programs measure the execution time as well as the auxiliary heap space usage. This will enable you to compare the performance and space 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. Incremental Milestone

We 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 (1) implement the list and (2) write an extensive test suite including many directed and random tests for this implementation.

6. 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 script that will autoformat the code for you. You can use it like this:

```
% cd ${HOME}/ece2400/netid/pa2-dstruct
% ./scripts/format.sh
% git diff
# ... check all changes ...
% git commit -a -m "autoformat"
```

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 `format.sh` script 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.

6.1. 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. Your entire report must be no more than four pages. You will have to use this Overleaf template to generate your pdf:

`https://tinyurl.com/2400-sp25-pa2temp`

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 in the Overleaf document). For time complexity analysis, you need to pick a key operator. For space complexity analysis, you need to analyze the *auxiliary 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 auxiliary heap space usage (see Fig. 1(a-c) in the Overleaf document). Create the plots using the data recorded from your quantitative evaluation. 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 polynomial equation as a function of N for each data series determined using a tool of your choice (see Table 2 in the Overleaf document). The equation should be in units of microseconds for execution time and in units of bytes for heap space usage. Use your complexity analysis to choose the most appropriate degree when doing your polynomial regression. If your complexity analysis suggests the measured data should be $O(1)$ then you should use a 0th degree polynomial fit (i.e., just take the average). If complexity analysis suggests the measured data should be $O(N)$ then you should use a 1st degree polynomial fit (i.e., just use linear regression). If complexity analysis suggests the measured data should be $O(N^2)$ then you should use a 2nd degree polynomial fit, and so on. You should also verify either qualitatively or quantitatively that this produces a good fit. **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 polynomial equations.**

Acknowledgments

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