1 Goals

- Review implementation of linked lists in `C++`;
- Practice using interfaces (abstract classes) and templates;
- Practice with unit tests and running and analyzing performance tests.

Note that you may use whatever environment you like for this class, but your programs must be able to compile and run using `g++` (or `clang++`), `cmake`, and `make`. For this assignment, you will also need `valgrind` to check for memory errors and leaks as well as `gnuplot` for generating performance graphs. You may also find it helpful to have `gdb` for helping to debug segmentation faults. The department provides a remote development server (`ada.gonzaga.edu`) running Ubuntu that can be accessed using an ssh remote connection from within VS Code on your own computer. The remote server contains all of the tools you need for assignments in this class. Depending on your computer's configuration, you may be able to install the tools you need locally to complete the homework assignments. It is important that you start assignments early in this class so that if you have questions you can ask them and get them answered with enough time before the homework deadline (to avoid late penalties).

2 Instructions

1. Use `git clone` to clone the classroom repository created for you and to obtain the starter code (either onto the remote development server if using `ada` or locally if you are using your own machine). Be sure to frequently add, commit, and push your updated files back to your GitHub repository (via `git add`, `git commit`, and `git push`).
2. Finish the implementation of `linkedseq.h`.
3. There are 20 total unit tests provided in `hw2_test.cpp`. For this assignment you do not need to write new unit tests, however, you will need to read through the tests provided and understand the purpose of each test.
4. You must run `valgrind` on the `hw2_test` program to ensure it does not detect any memory leaks in your `linkedseq.h` implementation. You should run `valgrind` once you get the unit tests to pass. It may also help if your program has memory issues that are causing segmentation faults or other errors. Similarly, `gdb` can also be helpful in these cases.
5. Once your code is completed, the unit tests pass, and you do not have memory issues as reported by `valgrind`, run the provided performance tests in `hw2_perf`. The performance tests will generate a data file with testing results, which you will then graph using `gnuplot`. You will need to add the corresponding graphs to your assignment write up (next step).
6. Create your assignment write up and add it as a PDF file to your assignment (GitHub) repository. You **must** save your writeup as a PDF file and name it `hw2-writeup.pdf`. (Note no capital letters, no spaces, etc.) Be sure to push all of your source code and your write up by the due date so it

can be graded. (Note that you can check that everything is in your repo from the GitHub website and/or using the `git status` command.) See below for expectations of concerning your assignment writeup. Once you have submitted your files to your GitHub repository and are ready to submit your assignment for grading, you must fill out the grading submission form. A link to the form will be provided in piazza (in the same post as the GitHub classroom link).

3 Additional Details

Maintain a head and tail pointer. For this assignment, your code *must* maintain both a `head` and `tail` pointer (see `linkedseq.h`). We will use the tail pointer specifically for the `insert` function as well as for accessing the last element of a sequence. Maintaining a `tail` pointer adds some extra cases to check for when adding, removing, and accessing nodes from the linked list.

Check for valid sequence indexes. You must check for valid indexes in your corresponding `LinkedSeq` functions. If an invalid index is given, a `std::out_of_range` error must be thrown. An invalid index is one that is not between 0 and $n-1$ (inclusive), except for the `insert` function, where an index of n is allowed (to insert at the end of the sequence).

Insert and `operator[]` should optimize the end-of-sequence case. Your `insert` function must take advantage of the tail pointer to optimize inserting at the end of a sequence. Similarly, accessing and updating sequence elements at the end of the linked list via `operator[]` should also directly use the tail pointer.

Implement all of the "essential operations". You must implement all six of the C++ essential operations: default (empty) constructor, copy constructor, move constructor, copy assignment, copy move, and a destructor. Note that your destructor simply needs to call the `clear()` function and the body of your default constructor should be empty.

Check for memory issues. As part of this and future assignments, a portion of your grade will be based on whether or not your code contains memory errors and/or memory leaks. You can use `valgrind` to check for memory errors and leaks (and sometimes to help find them). To run `valgrind` from the command line, pass it the `hw2_test` file as follows:

```
valgrind -s ./hw2_test
```

Note the `-s` flag tells valgrind to show a list of errors (if there are any). If valgrind does not detect any errors or leaks, you will see a message like this:

```
==84835==
==84835== HEAP SUMMARY:
==84835==     in use at exit: 0 bytes in 0 blocks
==84835==   total heap usage: 683 allocs, 683 frees, 143,936 bytes allocated
==84835==
==84835== All heap blocks were freed -- no leaks are possible
==84835==
==84835== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

The following is an example where valgrind found a memory leak (in this case, the body of the destructor was commented out):

```
==85161==
==85161== HEAP SUMMARY:
==85161==     in use at exit: 848 bytes in 53 blocks
==85161==   total heap usage: 683 allocs, 630 frees, 143,936 bytes allocated
==85161==
==85161== LEAK SUMMARY:
==85161==    definitely lost: 288 bytes in 18 blocks
==85161==    indirectly lost: 560 bytes in 35 blocks
==85161==      possibly lost: 0 bytes in 0 blocks
==85161==    still reachable: 0 bytes in 0 blocks
==85161==         suppressed: 0 bytes in 0 blocks
==85161== Rerun with --leak-check=full to see details of leaked memory
==85161==
==85161== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Note that in this case valgrind tells us we can rerun valgrind with `--leak-check-full` (e.g., `valgrind -s --leak-check=full ./hw2_test`). While sometimes useful, this additional option can often produce a significant amount of information that is hard to sift through. If valgrind detects memory errors, it will output the errors as it finds them while running your program. Things to look for in terms of errors include:

- *Invalid read/write of size X.* Your program tried to read from memory or store to memory X bytes of data in an invalid location (e.g., memory that was never allocated or already deleted).
- *Use of uninitialised value* or *Conditional jump or move depends on uninitialised value(s).* Your program is trying to access memory that was never initialized. The conditional jump typically implies the memory is being checked in an if, while, or for-loop comparison expression. Note that valgrind only reports this error if the unitialized address ends up causing a memory error. To help track down an unitialized value, you can try the `--track-origin=yes` flag.
- *Invalid free().* Your program attempted to delete a memory address that is not in the free store (i.e., not allocated on the heap), or else it is trying to delete the same address more than once.

Running `gdb`. If your program has a segmentation fault, you can use `gdb` to help track it down in terms of the location where the fault occurs. Note that the fault location is often not where the logic error occurs, but can provide useful information to help diagnose the issue. You can run `gdb` at the command line with `gdb hw2_test`. This command will put you into an interactive `gdb` shell. Here is an example:

```
bowers@laptop:~/cpsc223/src/hw2$ gdb hw2_test
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.

... removed output ...

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from hw2_test...
(gdb)
```

Assuming your program has a segmentation fault, at the prompt, type "`run`". Here is a snippet of the output.

```
(gdb) run
Starting program: /home/bowers/svn/bowers/teaching/cpsc223-s22/src/hw2/hw2_test
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[==========] Running 20 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 20 tests from BasicLinkedSeqTests
[ RUN      ] BasicLinkedSeqTests.EmptySeqSize
[       OK ] BasicLinkedSeqTests.EmptySeqSize (0 ms)

... removed output ...

Program received signal SIGSEGV, Segmentation fault.
0x00005555555e5769 in LinkedSeq<char>::insert (this=0x7fffffffdb40,
    elem=@0x7fffffffdad0: 98 'b', index=1)
    at /home/bowers/cpsc223/src/hw2/linkedseq.h:254
254     tail->next = new_node;
(gdb)
```

In this example, `gdb` is telling us that the segmentation fault occurred on line 254 of `linkedseq.h`, which was within a call to the `insert` function with parameters `'b'` and `1`. Sometimes it is useful to see the full stack trace that led up to the segmentation fault (especially if the segmentation fault is caused by a system call). To see the full stack trace use the `backtrace` command within `gdb`. Here is an example:

```
(gdb) backtrace
#0  0x00005555555e5769 in LinkedSeq<char>::insert (this=0x7fffffffdb40,
    elem=@0x7fffffffdad0: 98 'b', index=1)
    at /home/bowers/cpsc223/src/hw2/linkedseq.h:254
#1  0x00005555555de110 in BasicLinkedSeqTests_MoveConstructorChecks_Test::TestBody
    (this=0x555555685530) at /home/bowers/cpsc223/src/hw2/hw2_test.cpp:244
#2  0x0000555555627dc0 in void testing::internal::HandleSehExceptionsInMethodIfSupported<testing::Test,
    void>(testing::Test*, void (testing::Test::*)(), char const*) ()
#3  0x000055555561fc35 in void testing::internal::HandleExceptionsInMethodIfSupported<testing::Test,
    void>(testing::Test*, void (testing::Test::*)(), char const*) ()
#4  0x00005555555f46d8 in testing::Test::Run() ()

... output removed ...

#10 0x0000555555603d40 in testing::UnitTest::Run() ()
#11 0x00005555555e422e in RUN_ALL_TESTS ()
    at /usr/local/include/gtest/gtest.h:2497
#12 0x00005555555e22da in main (argc=1, argv=0x7fffffffe0d8)
    at /home/bowers/cpsc223/src/hw2/hw2_test.cpp:391
(gdb)
```

The output of backtrace can be read "bottom up" to see the sequence of calls made that led to a segmentation fault. In this case, `main` was called, then a number of google test functions, the `MoveConstructorChecks` at call `#1`, and finally the call to `insert` at `#0`. Note that in some cases it can also help to what led to the segmentation fault, e.g., in this case that the call on line 244 of `MoveConstructChecks` is what called the failing `insert`. Finally, to exit `gdb` type `quit` at the prompt (which may then ask you if you want to stop the process, which you will need to do to exit).

Homework Writeup. For this homework assignment, you will be generating three separate performance graphs, which will be generated for you via the provided `plot_script.gp` file. As in HW-1, your homework write up must contain each of the graphs together with an explanation as to why you think the performance tests came out the way they did based on what you know about the implementations. In addition, for HW-2, you must also provide a one sentence description of each provided unit test (in `hw2_test.cpp`). For example, for `AddAndCheckSize`, your description might be "*Inserts five different values into the sequence at different locations, and checks that the size and empty functions are correct after each insert.*" Finally, briefly describe any challenges or issues you faced in completing the assignment.