CSCE 121 Exam 3

Version 20.11.21-24

# Academic Integrity

Remember:

* Aggies do not lie, cheat, or steal, nor tolerate those who do.
* You know a lot of things and your computational thinking is evolving.
* We hope you have fun solving these problems and enjoy this celebration of learning.
* **You can do this!**

# Instructions

1. Read this exam prompt thoroughly and refresh it often during the exam period.
   1. You can use the document outline to more quickly move between parts of the exam.
      1. Make sure it is available. To check, in the menu go to “View->Show document outline”. Then you can click on the Outline Iconicon to the left near the top to open the outline.
2. Download the starter code from Mimir
   1. Do not change the file names. Otherwise, it will not be graded.
   2. File names
      1. LinkedList.h
      2. LinkedList.cpp
      3. Tests.cpp
3. Get started
   1. Think before you code
   2. Draw pictures to help you think
4. Compile early and often.
5. Test early and often.
6. [Submit to Mimir](https://class.mimir.io/projects/e0a76c27-6c03-47f2-b6ee-a48c00be4661) early and often.
7. Work alone
8. Ask for clarification on Piazza (private posts only)
   1. Refresh and read the [FAQ](https://docs.google.com/document/d/1uxiX04GwlykOvS166Z4wkoxzAYXInBKRNZtrqMAyxkk/edit?usp=sharing) first
9. [**And Just Have Fun!**](https://i.pinimg.com/originals/8f/bf/c5/8fbfc5980f207074aa4d6b0cfab6a116.jpg)

## 

## Allowed Includes

**Use of an “illegal” header file will result in a zero (0) on the exam.**

* **This is shown as a visible test cast on Mimir.**
  + **Failing the “Approved Includes” test case will result in a zero (0) on the exam.**
* **List of allowed Includes**
  + <cmath>
  + <iostream>
  + <istream>
  + <ostream>
  + <sstream>
  + <stdexcept>
  + <string>
  + “LinkedList.h”
  + “test\_helpers.h”
* If there is a library not on this list that you think you need, ask us on Piazza and we’ll consider it.

## How to Compile

**Failure to submit compilable code will result in a zero (0) on the exam.**

* **This is shown as a visible test case on Mimir.**
  + **Failing the “Compiles without errors” test case will result in a zero (0) on the exam.**
  + **Recall that warnings do not prevent code from compiling.**

g++ -std=c++17 -Wall -Wextra -pedantic-errors -g LinkedList.cpp Tests.cpp

## How to Use Tools for Checking for Memory Errors

### Sanitizers

g++ -std=c++17 -Wall -Wextra -pedantic-errors -g -fsanitize=address,undefined LinkedList.cpp Tests.cpp

./a.out

### Valgrind

g++ -std=c++17 -Wall -Wextra -pedantic-errors -g LinkedList.cpp Tests.cpp

valgrind --leak-check=full ./a.out

## Example Test Cases

Every problem on the exam has some provided example use case code, which is replicated as 0-point test cases on Mimir. There is no automatic determination for “passing” these tests.

* You will get a “50%” if your code compiles and runs.
  + Click on the test case to see the details.
* The tests output the contents of the list at certain points and you must decide whether the output is correct.
  + That is, the other 50% of the test case is in your mind.
* These are visible test cases on Mimir.
  + If you get 50%
    - You know that your code compiled and ran with the example test cases.
    - You DO NOT KNOW if your results are correct.
* They will never get to 100%. They are 50% (compiles and runs) or 0% (does not).

# Grading Overview

|  |  |
| --- | --- |
| Item | Points |
| [Code Coverage](#_mpmyxsdj2ohr) (bonus) | 10 |
| [Problem 0: Basic Linked List Infrastructure](#_q8fc01fj9qob) | 20 |
| [Problem 1: Remove all Duplicates from a Linked List](#_efij0wol8fdf) | 40 |
| [Problem 2: Length of Maximum Subsequence of Decreasing Values](#_oty5824bu6j8) | 20 |
| [Problem 3: Rule of Three](#_sznh728jvgc6) | 20 |
| Total: | **110** out of 100 |

# 

# Problem Theme: Linked List

You must implement a Linked List that supports the following:

1. basic linked list infrastructure
2. remove duplicate values from the list
3. find the length of the maximum subsequence of decreasing values
4. rule of three

## What is Provided Must Also Be Submitted

* LinkedList.h
  + struct Node defines the Node object type.   
    ***The provided types, names, and access modifiers must not be changed.***
    - int data
      * stores the data value.
    - Node\* next
      * stores the address of the next node in the list.
    - You may add to Node, but it is not required. The provided Node is sufficient.
  + class LinkedList defines the LinkedList object type.
    - The type to use for sizes and indices is std::size\_t.
    - You must implement the rest of LinkedList.
* LinkedList.cpp
  + *Nothing*
  + You must implement the rest of LinkedList.
* Tests.cpp
  + An empty main function stub.
  + You must write your tests in this file and use them in main.
  + Your tests *will* be executed and must run without error.
    - You should **NOT** comment out your tests, since your tests will be scored for [code coverage](#_mpmyxsdj2ohr).

# Code Coverage

## Overview

The code coverage assessment measures how much of your code your test cases exercise by being called from the main function in Tests.cpp.

* It does not care that you get correct results, it only counts how many times each executable line gets executed and reports the percentage of lines executed.
* You must have at least 80% coverage to pass the code coverage test. In practice, 100% coverage is often effectively unattainable so developers aim for “enough” coverage.
* This testcase is visible in Mimir.

## Points

* 10 bonus points
* You need at least 80% coverage.
* This is shown as a visible test case on Mimir.

# Problem 0: Basic Linked List Infrastructure

## Overview

You must design and implement any and all Linked List infrastructure that is necessary for the remaining problems, see the specification of [Required Functions](#_hwm74blqq9ck) below.

## Points

* 20 points
  + Partial credit is available

## Example

LinkedList list;

*// list := <empty>*

list.push\_back(8);

list.push\_back(6);

list.push\_back(7);

list.push\_back(5);

list.push\_back(3);

list.push\_back(0);

list.push\_back(9);

*// list := 8, 6, 7, 5, 3, 0, 9*

## Required Functions

*You are required to provide these public functions.You may add additional public and private helper functions that support the required functions.*

### LinkedList::LinkedList()

The LinkedList default constructor makes an empty list (zero size, null head).

#### Parameters

None.

#### Return Value

None.

#### Exceptions

None.

### std::size\_t LinkedList::size() const

Return the number of elements in the list.

#### Parameters

None.

#### Return Value

The number of elements in the list.

#### Exceptions

None.

### const Node\* LinkedList::head() const

Return the pointer to the head of the list.

* *Note:* ***This is for testing purposes only!*** *You would normally not make a public head function since the public interface should protect the inner workings of your class.*

#### Parameters

None.

#### Return Value

Pointer to the head of the list, or nullptr if the list is empty.

#### Exceptions

None.

### void LinkedList::push\_back(int value)

Append the given value to the end of the list.

#### Parameters

value - the value to append.

#### Return Value

None.

#### Exceptions

None.

# Problem 1: Remove all Duplicates from a Linked List

## Overview

You must implement a function that removes duplicate values from the list. You must design and implement any and all Linked List infrastructure that is necessary for this operation, see the specification of [Required Functions](#_4uxd3qw76o7d) below.

Removing duplicates from a list means transforming the list such that each value in the list appears exactly once.

## Points

* 40 points
  + Partial credit is available

## Examples

LinkedList list;

*// list := <empty>*

list.push\_back(1);

list.push\_back(7);

list.push\_back(2);

list.push\_back(7);

list.push\_back(7);

list.push\_back(3);

list.push\_back(7);

list.push\_back(7);

list.push\_back(4);

list.push\_back(7);

list.push\_back(5);

*// list := 1, 7, 2, 7, 7, 3, 7, 7, 4, 7, 5*

list.remove\_duplicates();

*// // the following and all permutations thereof are correct:*

*// list := 1, 2, 3, 4, 5, 7*

The following example is **not** a visible test on Mimir.

LinkedList list;

*// list := <empty>*

list.push\_back(1);

list.push\_back(2);

list.push\_back(3);

list.push\_back(4);

list.push\_back(3);

list.push\_back(2);

list.push\_back(1);

*// list := 1, 2, 3, 4, 3, 2, 1*

list.remove\_duplicates();

*// the following and all permutations thereof are correct:*

*// list := 1, 2, 3, 4*

## Required Functions

### *You are required to provide these public functions. You may add additional public and private helper functions that support the required functions.*

### void LinkedList::remove\_duplicates()

Remove all duplicate values from the list, keeping only a single element of each distinct value in the list. The order of the list after removing all duplicates is unspecified (i.e. order is not necessarily preserved).

#### Parameters

None.

#### Return Value

None.

#### Exceptions

None.

## Optional (Partial Credit) Functions

If you are unsure of how to solve this problem, you may refer to [Appendix: Partial Credit Methods](#_hfed1q7bk8p2) which lists some methods which you can implement for partial credit in lieu of a correct and complete implementation of remove\_duplicates.

# Problem 2: Length of Maximum Subsequence of Decreasing Values

## Overview

You must implement a function that returns the length of the maximum subsequence of strictly decreasing values. See the specification of [Required Functions](#_ndm4uktyqrur) below.

## Points

* 20 points
  + Partial credit is available

## Example

LinkedList list;

*// list := <empty>*

list.push\_back(11);

list.push\_back(9);

list.push\_back(3);

list.push\_back(8);

list.push\_back(7);

list.push\_back(5);

list.push\_back(2);

list.push\_back(1);

list.push\_back(4);

list.push\_back(3);

list.push\_back(9);

list.push\_back(8);

*// list := 11, 9, 3, 8, 7, 5, 2, 1, 4, 3, 9, 8*

std::size\_t max\_length = list.length\_max\_decreasing();

*// max\_length will be 5*

The following examples are **not** visible tests on Mimir.

LinkedList list2;

*// list := <empty>*

list2.push\_back(11);

*// list := 11*

std::size\_t max\_length2 = list2.length\_max\_decreasing();

*// max\_length will be 1*

LinkedList list3;

*// list := <empty>*

list3.push\_back(11);

list3.push\_back(4);

*// list := 11, 4*

std::size\_t max\_length3 = list3.length\_max\_decreasing();

*// max\_length will be 2*

LinkedList list4;

*// list := <empty>*

list4.push\_back(11);

list4.push\_back(11);

*// list := 11, 11*

std::size\_t max\_length4 = list4.length\_max\_decreasing();

*// max\_length will be 1*

## Required Functions

### *You are required to provide these public functions. You may add additional public and private helper functions that support the required functions.*

### std::size\_t LinkedList::length\_max\_decreasing() const

Determine the length of the maximum subsequence of decreasing values.

#### Parameters

None.

#### Return Value

The number of elements in the longest contiguous subsequence of strictly decreasing values in the list, or 0 if the list is empty.

#### Exceptions

None.

# Problem 3: Rule of Three

## Overview

Implement the rule of three for the Linked List class. See the specification of [Required Functions](#_wbuhggjihvwe) below.

## Points

* 20 points
  + Partial credit is available

## Example

LinkedList list\_A;

// list\_A := <empty>

for (**int** n = 1; n <= 3; n++) {

list\_A.push\_back(n);

}

// list\_A := 1, 2, 3

{

LinkedList list\_B(list\_A);

// list\_A := 1, 2, 3

// list\_B := 1, 2, 3

list\_B.push\_back(4);

// list\_A := 1, 2, 3

// list\_B := 1, 2, 3, 4

list\_A = list\_B;

// list\_A := 1, 2, 3, 4

// list\_B := 1, 2, 3, 4

list\_B.push\_back(5);

// list\_A := 1, 2, 3, 4

// list\_B := 1, 2, 3, 4, 5

}

list\_A.push\_back(5);

// list\_A := 1, 2, 3, 4, 5

## Required Functions

*You are required to provide these public functions. You may add additional public and private helper functions that support the required functions.*

You must identify and implement any functions needed for the rule of three.

# 

# Appendix: Partial Credit Methods

## Partial Credit Points

* 20 points
  + Partial credit is available

You are **not** required to provide these functions, but you may do so to earn partial credit in the absence of a correct and complete implementation of remove\_duplicates.

**To Be Clear: NONE of these functions is required. There are many ways to solve this problem, some of which do not use any of these functions. One or more of these functions may be helpful to you, either for solving the problem or for identifying mistakes in list management. Just because you see a function listed below does not mean that it is “The Way™” or any part of “The Way™”.**

### bool LinkedList::contains(int value) const

Returns whether the list contains the specified value.

#### Parameters

value - the value for which to search.

#### Return Value

true if the list contains the specified value, false otherwise.

#### Exceptions

None.

### std::size\_t LinkedList::find\_last\_of(int value) const

Finds the last element equal to the given value. Use 0-indexing.

#### Parameters

value - the value for which to search.

#### Return Value

Position (index) of the found value or static\_cast<std::size\_t>(-1) if no such value is found.

#### Exceptions

None.

### void LinkedList::remove(std::size\_t index)

Removes the value at the specified index from the list. Use 0-indexing.

#### Parameters

index - index of the value to remove.

#### Return Value

None.

#### Exceptions

std::out\_of\_range if index >= size().