Final Exam Review - Fall 2023

Exam Details

- The exam is made up of:
    - Writing code to solve problems.
    - Tracing execution given specified input values.
- There are multiple versions of the exam.
- **Partial credit is available** for every question.
    - Your goal should be to not need it.
    - Focus your solution on correctness of the problem solving method. Clear evidence of your thought process and plan must be included in the code and comments in order to qualify for partial credit.
    - Put down anything you do know even if you can't solve the entire problem. Some partial credit is better than getting no credit.
    - Make your thinking visible so applying partial credit is easier.
- **Not allowed**
    - Any electronic devices (including calculators, phones, smart watches, and computers)
- **Allowed**
    - A writing utensil (e.g. pen/pencil)
    - Scratch Paper (8.5X11 inches)
    - Up to 5 pages of exam aids that you create.
        - Exam aids can be pages up to 8.5X11 inches and can be handwritten or printed on both sides.
        - You can't use exam aids for scratch work during the exam.
            - If you do, then you must submit the exam aid with your exam.
- We will be manually grading the exam, so don't get stressed if it is not perfect since you will get partial credit. We will also ignore minor errors that do not significantly impact the results.

Exam Strategies

- Understanding at a high level will help you code better during an exam. So, as you study, ask yourself…
    - *What is happening?*
        - *High level: What is the goal?*

        - *Code level: What are the effects and consequences of statements, etc.?*
    - *Why is it that way?*
        - *High level: Why are certain design decisions made? Why is it designed that way?*
        - *Code level: What is the code that accomplishes a higher-level goal? Why is it done that way?*
    - *How is it done?*
        - *High level: How do you accomplish a high-level goal?*
        - *Code level: How do you write the code?*
- Study Strategy: Prioritize how you study to maximize your ability to demonstrate concept mastery while not overwhelming yourself.
    - For a solid passing grade focus on class topics, slides, examples, in-class activities, homework, and labwork.
        - Make sure you understand anything coded in homework, labwork, and in-class examples.
            - What general (higher level) principles were illustrated in a labwork/homework?
                - Could you explain it in general?
                  (i.e. what/why rather than how)
                - Be able to solve and write code for similar problems.
    - After you've done all you can do in the "for a solid passing grade" above, then focus on zyBook Challenge Activities and Participation Activities.
        - If a challenge or activity seems fuzzy, review the text for clarification.
        - Key terms are bolded.
    - After you've done all you can from the previous approaches and if you are still fuzzy about a topic, read about it in a different source or textbook. You can also talk to someone who can explain it in a different way.

Note, topics are not always organized sequentially!

Exam Topics

Programming is inherently cumulative. So all concepts covered can show up as parts of any question. So, looking over prior review sheets should remind you of concepts that may not have been covered explicitly on prior exams.

Since this is the Final Exam, some topics from previous exam materials may be the focus of some questions.

Overarching Themes

- Software Development Process
  - Design
    - Flowcharts
    - Pseudocode
    - Creating a Model
- Incremental programming
- Programming Goals
- Memory Diagram
  - Variables
  - Functions
  - Recursion
  - Classes/Objects
  - Dynamic Memory
- Code Organization
  - Functions
  - Classes

Recursion

- Recursion vs Looping
- Base case
- Recursive cases

Objects and Classes

- Syntax
- Structure
  - Data members / attributes
  - Member functions / methods
  - Private helper functions
  - Constructor
    - Member initialization
    - Overloading
    - Default constructor
- Access
  - public / private / protected
  - Getters and Setters / Readers and Writers / Accessors
- class vs. struct
- class vs. object
- Operator overloading
- 'this'

UML

- Structure / Sections
- Relationships
  - Aggregation
  - Composition
  - Inheritance
  - Etc.
- Is-a (Inheritance) vs. has-a (Composition) relationships
- Public / Private

Dynamic Memory

- Operations
  - Allocation
  - Deallocation
  - Access
- Where allocated
- Memory Management
  - Garbage Collection
  - Memory Leaks
    - Definition
    - Prevention strategies
      - Especially RAII
- Shallow vs. Deep Copy
- Class Design
- Rule of Three (Signatures and Why needed?)
  - Destructor
  - Copy Constructor
  - Copy Assignment Operator

Linked List

- Singly and Doubly linked lists
- Structure
  - Head
  - [Tail]
  - Node Structure
- Know how to do basic operations for both types
  - Traversal
  - Insert / Remove
    - Front
    - Back
    - Middle

Inheritance and Polymorphism

- Pros / Cons
- Class hierarchies
- Base class / superclass / parent class
- Derived class / subclass / child class
- Access
  - Public / private / protected
  - Data members / Member functions / Constructors
- const Member functions
- Overriding methods
  - Virtual Functions
- Abstract Classes (pure virtual)
  - Pure Virtual Functions
- When to use Virtual and Pure Virtual functions.
- Types of polymorphism
  - Ad-Hoc - function overloading
  - Subtyping - using derived class in place of base class
  - Parametric - templates
- How do you create a vector (or list, etc.) that holds elements of an abstract class?
- How do you create a vector (or list, etc.) that holds elements that all share the same base class?

Practice Problems

Note that practice problems are generally greater than or equal in difficulty to what you will find on the exam. Exam questions are designed to be challenging and doable in the time provided. Some of the practice questions are longer than what you would see on an exam, but should give you good practice.

Don't forget to review practice problems from review sheets for prior exams.

For linked list questions assume the following Linked list class definition:

```
struct Node {
  int value;

  Node* next;
  Node(int num = 0) : value{num}, next{nullptr} {}
};


class LinkedList {
  Node* head;


 public:
  LinkedList() : head{nullptr} {}
  // other member functions
};
```

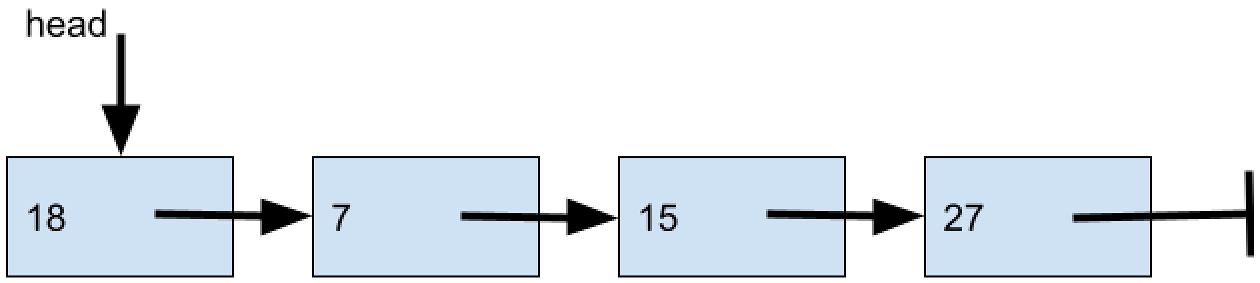
Linked List Average

Write a member function of the class LinkedList (a simply-linked list, with a pointer head to its first element) that returns the average of the elements in the list.

- Do not assume the existence of any other functions to use.
- Throw an exception if the list is empty.

Example

For the list



the function should return 16.75.

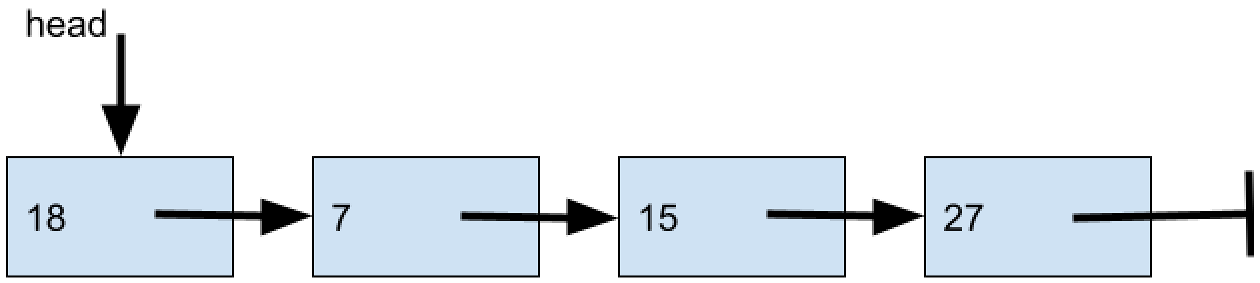
Remove all Maximum

Write a member function of the class LinkedList that removes from the list all occurrences of the maximum element in the list.

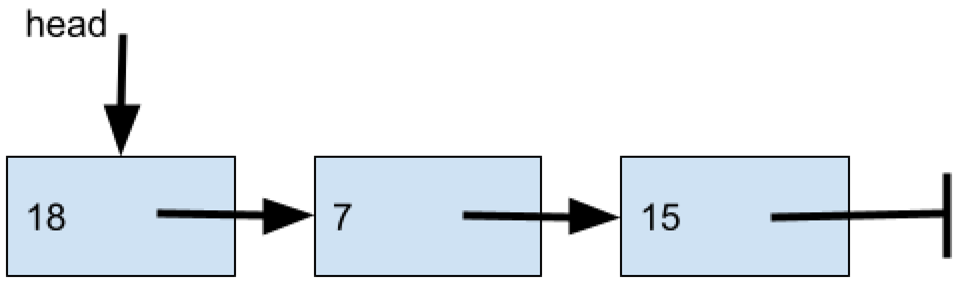
- Observe that the list can be and can become empty.
- Do not assume the existence of any other functions to use.

Example 1

The list

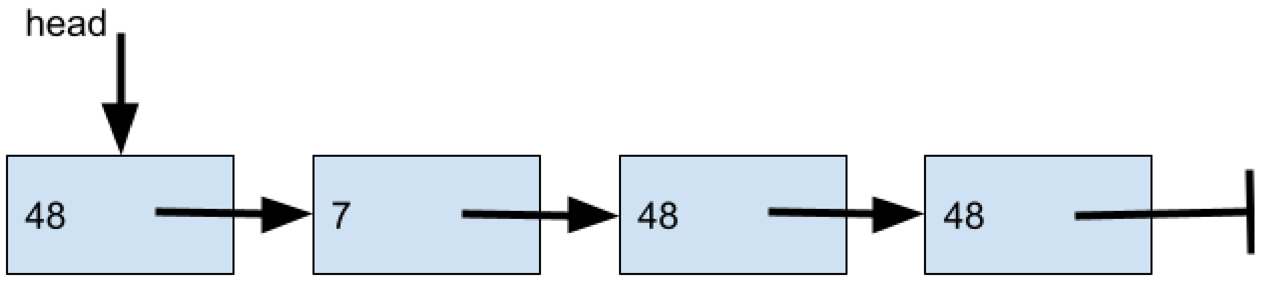


Will become

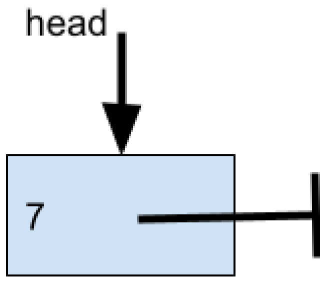


Example 2

The list



Will become



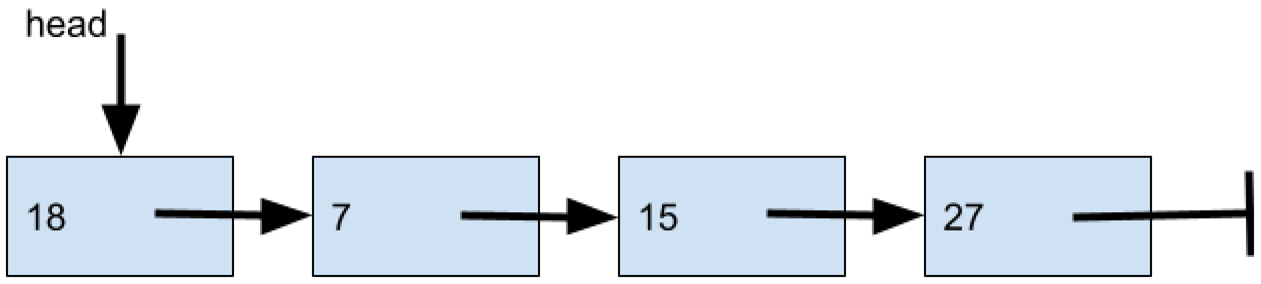
Reverse Linked List

Write a member function of the class LinkedList that reverses the list.

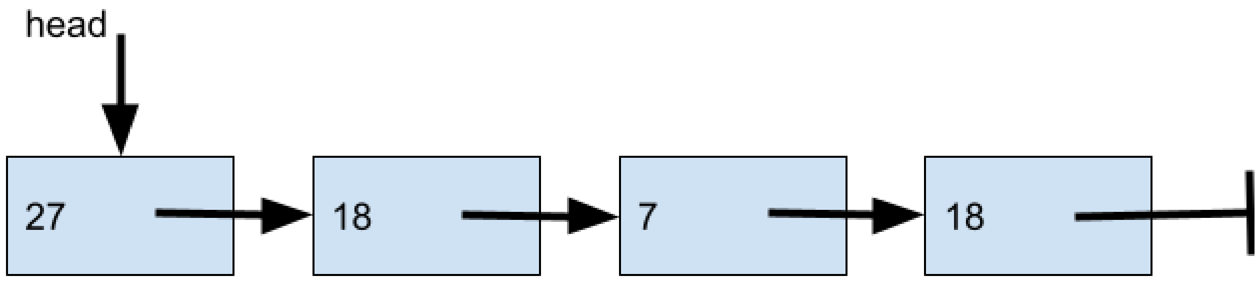
- Observe that the list can be empty.
- Do not assume the existence of any other functions to use.

Example

The list


head
18
7
15
27

Will become



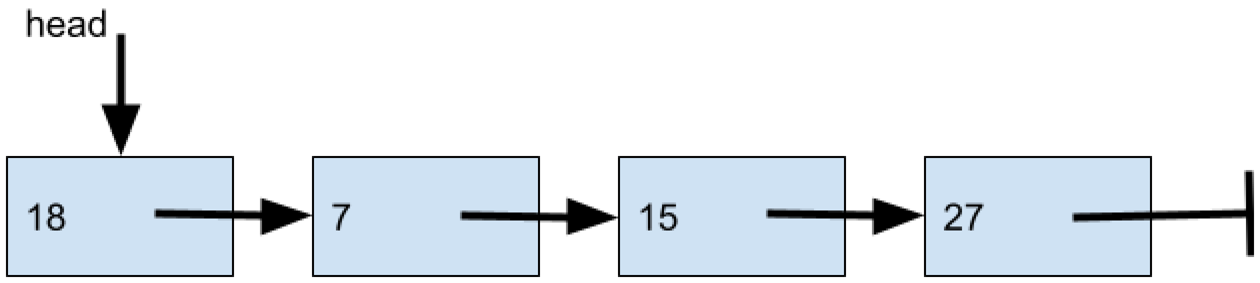
Middle of the List

Write a member function of the class LinkedList that returns a pointer to the element in the middle of the list (if the list has n elements, it should return a pointer to the $(\left\lfloor \frac{n}{2} \right\rfloor + 1) - th$ element).

- Do not assume the existence of any other functions to use.
- Return nullptr if the list is empty.

Example

For the list



The function should return a pointer to the node that contains 15.

Linked List Cycle

Write a member function of the class LinkedList that returns true if a cycle exists in the linked list, false otherwise.

- Do not assume the existence of any other functions to use.
- Return false if the list is empty.

What does “cycle” mean? The singly linked list is a linear sequence. That is, starting with the *head*, if we traverse each node, no node will be visited twice. However, if the *next* pointer stored within a node points to an already traversed node, it will form a cycle (see example below).

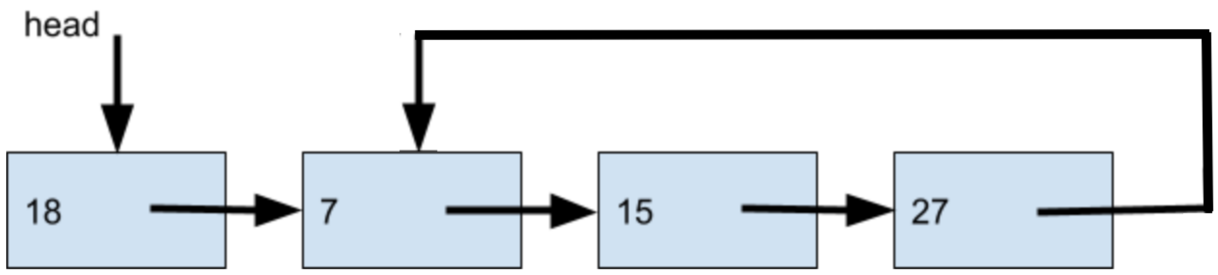
Hint

You need to maintain information about the traversed/visited nodes. A couple of options:

- [bad] Use a bool variable in the Node class (violates open-closed principle)
- Use a collection (e.g., dynamic array or vector) to store the information about visited nodes (What information? What if values are not unique?)
- Use Floyd's tortoise and hare: traverse the list with two pointers, the tortoise that goes in steps of 1 node, and the hare that goes in steps of 2 nodes.

Example

For the list



The function should return true.

Polygon Class (Rule of Three)

A polygon is a shape in the Cartesian plane that is defined by an ordered list of points. The edges of the polygon are the line segments whose endpoints are adjacent to each other in the list, as well as the line segment between the first and last points.

We will represent points using the struct Point, each instance of which has an x- and a y-coordinate as its two data members. An instance of the Polygon class will have two data members: a pointer to an array of Point objects and an int representing the length of the array. Implement the member functions for the Polygon class, including the Rule of Three: a destructor, a copy constructor, and a copy assignment operator.

```
struct Point {
  int x;
  int y;
};

class Polygon {

private:
  Point* vertices; // remember to think arrays for this
  int numVertices;


public:
  Polygon(); // constructor
  ~Polygon(); // destructor
  Polygon(const Polygon& poly);  // copy constructor
  Polygon& operator=(const Polygon& poly);  // copy assignment
operator

  // other member functions
  void addVertex(int x, int y);
  Point* getVertices() const;
  int getNumVertices() const;
  void print(); // outputs the list of vertices (x1, y1), (x2, y2),
...
};

int main() {
  Polygon p1;
  p1.addVertex(1, 2);
  p1.addVertex(2, 5);
  p1.addVertex(3, 4);
  p1.addVertex(3, 3);

  Polygon p2(p1);
  p2.addVertex(3, 2);

  Polygon p3;
  p3 = p1;
  p3.addVertex(2, 2);
}
```

Polygon Class (Polymorphism and Inheritance)

Subclass Polygon with several specializations of polygons, e.g. rectangle, square, circle, star, pentagon, etc.

- Implement a single method that computers the perimeter of a polygon.
- Implement a single method that computes the area of any simple polygon.
    - See Shoelace formula

Palindrome (Recursion)

Given an integer, write a recursive function that returns true if the given number is palindrome, else false.

Examples

- For the integer 12321, the function will return true (palindrome)
- For the integer 1451, the function will return false (not a palindrome)