CptS 122 – Data Structures

Final Exam Review Guide

This document will serve as a guide to help you prepare for the final written exam in CptS 122. You will find information about the exam format and topics you are expected to review within this guide.

What to Bring?

- Your WSU ID
- Two sharp pencils
- Calculators and other notes may **not** be used during the exam!

Exam Timeframe

The final exam is scheduled for **TH, July 25, 2024, 10:00 am – 12:00 pm**. It will be held in our normally scheduled classroom location. Note that, when you hand in your exam, you will be required to present your WSU ID to the exam proctor.

Exam Format

Expect the final exam to look a lot like the first exam, except that it will be longer, because **you will have a full two hours** to take the exam, rather than one hour. There will be a mix, of true-false, fill-in-the-blank, multiple-choice, and short answer/code questions that test your knowledge of key concepts. Expect also to supply short code snippets, and to trace through C/C++ code segments and specify their output.

Exam Coverage

The exam is comprehensive, covering all the material we have explored in this course.

Topics that are fair game from exam #1:

- See the midterm #1 exam review for a list of topics that are fair game. In other words, all of the material covered in the first three weeks of the course is fair game.

The following is a list of exam topics covered in the final five weeks of the course:

C++ Data Structures

- Define what is a dynamic data structure in C++
- Apply *new* and *delete* to dynamic data structures
- Compare and contrast linked lists, stacks, queues, and BSTs
- Compare and contrast *container* and *value* classes

- Design and implement an ordered or non-ordered dynamic linked list using C++ classes including the following methods:
  - isEmpty ( ) - returns an integer or bool type; true for an empty list, false for non-empty list
  - insertAtFront ( ) – allocates a node dynamically; initializes it to the data passed in; inserts the node at the front of the list only; returns true or false for successful or unsuccessful insertion, respectively
  - insertAtEnd ( ) - allocates a node dynamically; initializes it to the data passed in; inserts the node at the tail or end of the list only; returns true or false for successful or unsuccessful insertion, respectively
  - insertInOrder ( ) - allocates a node dynamically; initializes it to the data passed in; inserts the node in the list in ascending or descending order only; returns true or false for successful or unsuccessful insertion, respectively
  - deleteNode ( ) – de-allocates a node dynamically; returns true if node was de-allocated, false otherwise
  - printList ( ) – prints out the data in each node of the list; may be printed iteratively or recursively
  - others?

- Design and implement a dynamic linked stack (LIFO – last-in, first-out) using C++ classes including the following methods:
  - isEmpty ( ) - returns an integer or enumerated bool type; true for an empty stack, false for non-empty stack
  - push ( ) - allocates a node dynamically; initializes it to the data passed in; inserts the node at the top of the stack only; returns true or false for successful or unsuccessful insertion, respectively
  - pop ( ) - de-allocates a node at the top of the stack dynamically; returns true if node was de-allocated, false otherwise; NOTE: some variations of pop ( ) will return the data in the node found at the top of the stack, instead of true or false
  - top ( ) or peek ( ) – returns the data found in the top node of the stack; nodes are not affected (removed)
  - printStack ( ) - prints out the data in each node of the stack; may be printed iteratively or recursively

- Design and implement a dynamic linked queue (FIFO – first-in, first-out) using C++ classes including the following methods:
  - isEmpty ( ) - returns an integer or enumerated bool type; true for an empty queue, false for non-empty queue
  - enqueue ( ) - allocates a node dynamically; initializes it to the data passed in; inserts the node at the tail/e of the queue only; returns true or false for successful or unsuccessful insertion, respectively
  - dequeue ( ) - de-allocates a node at the head/front of the queue dynamically; returns the data in the node found at the head/front of the queue; NOTE: some implementations may also return true or false for successful or unsuccessful removal of a node from the head/front

- printQueue ( ) - prints out the data in each node of the queue; may be printed iteratively or recursively

- Design and implement a dynamic linked binary search tree (BST) using C++ classes including the following methods:
  - isEmpty ( ) - returns an integer or enumerated bool type; true for an empty BST, false for non-empty BST
  - insert ( ) – allocates a node dynamically; initializes it to the data passed in; inserts the node into the left or right subtree; returns true or false for successful or unsuccessful insertion, respectively
  - inOrder ( ) – performs an inorder traversal of a BST and prints out the data in the nodes accordingly
  - preOrder ( ) – performs a preorder traversal of a BST and prints out the data in the nodes accordingly
  - postOrder ( ) - performs a postorder traversal of a BST and prints out the data in the nodes accordingly
  - destroyTree () – removes all nodes in the tree

- Design and implement makeNode ( ) as a separate helper function for each of the above data structures
  - makeNode ( ) – allocates a node dynamically; initializes the node; returns a pointer to the dynamic node
- How would you make use of private and public member functions for each of the container classes? Could you design a public preorder () function, which calls upon a private preorder () helper function? Etc.
- Draw block/memory diagrams to illustrate how links are modified for any of the particular operations described above
- Design and implement a list, stack, and queue with arrays instead of dynamic "links"

Chapters 1 - 3: Introduction to Classes and Objects

- Design and implement classes in C++
  - What are some advantages to using classes?
- Design and apply data members and member functions for classes
- Define and apply *accessor* (getter) functions and *mutator* (setter) functions
- Define *access specifier*
  - These include private, protected, and public
- Apply UML Class Diagrams
- Apply and implement default constructors, copy constructors, and destructors
- How is the size (amount of memory) of an object determined?
  - We generally assume an object contains data and operations…However, each instance of an object uses the same copy of the member functions, which is separate from the object size
  - Sizeof reports only the number of bytes required for a class's data members
- What is the rule of three/Law of the Big Three/the Big Three?
  - Rule of thumb in C++: should define destructor, copy constructor, and overloaded copy assignment operator

Chapter 6 & 9: Classes: A Deeper Look

- Compare and contrast procedural programming (C) versus objected-oriented programming (C++)
- Define the term *encapsulation*
  - Wrapping of attributes and operations into objects
- Define the term *information hiding*
  - Implementation details are hidden within objects
- Define and apply *function overloading*
  - Allows for functions with the same name to be defined. The key is the functions must have different parameters (number, type, order)
- Define and apply *procedural abstraction*
- Define and apply *data abstraction*
- What is a data member (attribute)? What is a member function (operation)?
- Define and apply function *templates*
- Apply the reference (&) operator in C++, including returning references from functions
- Define *pass-by-reference* and *pass-by-value*
- What is a *dangling* reference?
- Define, implement, and apply *friend* functions and classes
  - Recall: a friend class has access to private members
- What is the *this* pointer? When do we need to use it?
- Apply dynamic memory management with *new* and *delete* operators
- What is a *const object* and *const member function*?
- What is class *composition*?
  - Recall: represents a "has-a" relationship

Chapter 10: Operator Overloading

- List the operators that may not be overloaded (there are four of them…)
  - Recall: precedence, associativity, and number of operands (arity) for an operator may not be changed
- Implement and apply overloaded stream insertion and extraction operators
- Implement and apply overloaded unary operators
- Implement and apply overloaded binary operators (+, -, *, /, etc.)
- Compare and contrast overloaded member operators versus non-member operators
- Define what is a *forward class declaration*

Chapter 11: Object-Oriented Programming: Inheritance

- What is inheritance? When should we apply it?
  - Recall inheritance applies a "is-a" relationship
- Define, implement, and apply *base* and *derived* classes
  - Other terms include: subclass, superclass
- Describe when to apply the *protected* access specifier
- Describe the different inheritance access specifiers of C++ (i.e. public, protected, and private)
- What is *single, multiple, hierarchical, multilevel,* and *hybrid* inheritance?
- Describe the *diamond* problem

Chapter 12: Polymorphism

- Define the term *polymorphism*
- What is a *virtual* function? What is a *pure* virtual function?
- Provide an example of when/how polymorphism should be applied
- Define *abstract* class and *concrete* class
- What is a virtual function table or *vtable*?
- Implement and apply polymorphism
- How does polymorphism apply to computer game creation?

Chapter 17: Exception Handling

- Define what is an *exception*
- Implement and apply exception handling to C++ programs
- List and identify standard library exception classes (i.e. logic_error, runtime_error, etc.)
- Discuss when to apply exception handling

Chapters 18 & 19: Templates

- What is a *function* template?
- What is a *class* template?
- Design, implement, and apply *templates*
- What are advantages and disadvantages of templates?

Chapter 20: Searching and Sorting

- What is Big-O notation?
- What is the meaning of constant, linear, and quadratic runtime?
- Identify the runtimes for linear and binary search, and insertion, selection, and bubble sorts in the best, average, and worst-case
- Identify the runtimes for operations applied to lists, stacks, queues, and BSTs; these include both array and linked implementations of these data structures; what is the runtime for insertFront (), insertEnd (), insertInOrder (), deleteFront (), deleteEnd (), deleteN (), etc.

Other Material

- What is a function call stack?
- Apply recursion to a given set of problems, including BSTs
- What is a *stream*? What is a *file* stream?
- Open, read from/write to, and close files in C++
- What is *UML*?
- When should we *apply* UML diagrams?
- Construct and apply UML *class* diagrams
- What is the Standard Template Library (STL)? (Chapter 15.1 – 15.5)

Recommended Strategy for Preparing for the Exam

I recommend that you use the following activities and materials to prepare for the exam:

- **Review quizzes and lab exercises**: These may well be your best resource. An excellent learning activity would be to retake the quizzes and review the lab exercises.
- **Lecture slides and example code**: Study the lecture slides and example code. Continue to complete extra coding examples on your own time.
- **Read the textbook:** Read or re-read chapters 1 – 3, 6, 7, 9 - 12, 14, 15, 17 - 19, and 20 in your textbook. Solve the end-of-chapter exercises.