CS 132 Final Exam - Study Suggestions

* **last modified: 5-5-04**

* SOMETHING NEW: for the final, you may bring **a single 8.5" by 11" sheet with your name prominently on it** on which you have **handwritten** the course material of your choice, along with your name --- this sheet **must be turned in with your final exam**, and will not be returned. (You may write on both the front and the back, if you wish; and you must **personally** create/hand-write the sheet.)

* final is CUMULATIVE!
  * if it was fair game for exams #1 or #2, it is fair game for the final;
  * so, studying the review suggestions for exams #1, #2 would be wise; (they're still available from the course web page, under "Homeworks and Handouts", if you've misplaced your earlier copies);
  * studying the exams #1 and #2 themselves would also be wise;
  * there may indeed be similar styles of questions as on those exams;

* general style of questions will be similar to those on exams #1 and #2;

* Anything that has been covered in **assigned reading** is fair game;
* Anything that has been covered in **lecture** is fair game;
* Anything covered in a **course handout** is fair game;
* Anything covered in a **lab exercise** or **homework assignment** is **ESPECIALLY** fair game.

* The exam will be closed-book and closed-notes (except for the one 8.5" sheet mentioned above), and you are expected to work individually.
* Test format: will likely be short answer, possibly with a smattering of multiple-choice questions.
  * All you need to provide is a pen or a pencil (and the 8.5" sheet mentioned above, if desired).

  * EXPECT to have to read and write C++ code, pseudocode, pseudo-UML notation.

* You are responsible for being familiar with, and following, the class **style** guidelines.

* You should still be comfortable with the design recipe we've been using for functions, and should be able to fill in the opening comment block "templates" we've been using appropriately.

* the only aspect of namespaces that you are responsible for on this exam is that you need to use `using namespace std;` after #include'ing standard libraries in modern, standard C++.

* there could certainly be questions involving big-O notation; for example, I might ask what the big-O notation would be for the {average, best, worst} case run-time complexity for an instance of a particular ADT containing **n** items implemented in a particular way.

* You will likely be given pseudo-UML's for different typical container classes and/or template classes; you could be asked to write code **using** instances of such classes, and you could be asked to write **implementations** of member functions (or possibly even the entire class) given within such pseudo-UML's.

* high points from material SINCE Exam #2:
  * These are some especially-significant topics to help you in your studying for the exam.

* note that all of the **sorting algorithms** discussed --- including **treesort** and **heapsort** --- are most certainly fair game on this final! I want you to be able to compare/contrast/reason about/recognize all of them;

* you will have to read and write code involving pointers; expect it.

* **trees** continued
    * how might a binary tree be implemented using arrays? using linked nodes?
        * how might a complete binary tree, in particular, be implemented using an array?
    * some additional facts about binary trees that you should know (along with those that were fair game for Exam #2):
        * A **full** binary tree of depth **d >= 0** must have how many nodes?
        * The maximum number of nodes in a binary tree of depth d is what?
        * The minimum depth of a binary tree with n nodes is what?
        * what kinds of binary trees have this minimum depth?

* **priority queues** and **heaps**
    * what is a priority queue? how does a priority queue differ from a "regular" queue?
    * what are some typical operations defined on priority queues? how can a priority queue be used?
    * in what kinds of situations is a priority queue appropriate?
    * what are some typical applications of priority queues?
    * need to be able to **use** a given priority queue ADT or UML or interface to solve problems;
    * what is a **heap**? What **two** properties must a heap satisfy?
    * why does a **heap** work nicely for implementing a priority queue?
    * in what kinds of situations is a heap appropriate?
    * what are some typical applications of heaps?
    * how can a heap be implemented? Need to familiar with both linked and array-based implementations of a heap, but I expect you to be **more** familiar with the array-based implementation, since you did more with it in homeworks;
        * you should be able to compare/contrast these implementations; discuss their tradeoffs, big (O) complexity for different operations using the different implementations, etc.
        * Our definition of a heap assumes that it is **balanced**, remember. How can you insert into a complete heap, and ensure that it is still complete (and still a heap) after the insertion? How can you delete from a complete heap, and ensure that it is still complete (and still a heap) after the deletion?
    * need to be able to **use** a given heap ADT or UML or interface to solve problems; you also need to be able to implement (and reason about the big O complexity of) heap operations in the

different heap implementations.

* should be comfortable with **heapsort**; what is its average and worst-case performance complexity? you may have to give pseudocode for it, show and/or demonstrate its basic actions, or reason about it/recognize it on the final as well.

* **binary search trees**
  * what is a binary search tree? Is every binary tree a binary search tree? Is every binary search tree a binary tree?

  * you should be comfortable with **searching** a binary search tree. What are the average- and worst-case complexities for such searches?
    * how does the **shape** of the tree affect the performance complexity of binary search tree search?

  * you should be comfortable with **inserting into** a binary search tree. What are the average- and worst-case complexity for such insertions?
  * you should be comfortable with **deleting** from a binary search tree.

  * what are some of the typical operations defined on binary search trees? how can a binary search tree be used?

  * in what kinds of situations is a binary search tree appropriate?
  * what are some typical applications of binary search trees?

  * need to be able to **use** a given binary search tree ADT or UML or interface to solve problems; you also need to be able to implement (and reason about the big O complexity of) binary search tree operations in the different implementations.
    * you should be able to compare/contrast these implementations; discuss their tradeoffs, big (O) complexity for different operations using the different implementations, etc.

  * if you traverse a binary search tree inorder, printing the nodes' search key values as they are visited, what will be the case?

  * should be comfortable with the NON-REBALANCING **treesort**; what is its average and worst-case performance complexity? you may have to give pseudocode for it, show and/or demonstrate its basic actions, or reason about it/recognize it on the final as well.

* **hashes, hash tables, and hashing**
  * what is a hash function? What is a hash table? What is hashing?

  * what does a hash function do? How is it used?
  * what are some of the desired properties for a hash function? you should be able to implement a simple hash function, and be able to assess its quality;
    * what are some examples of hash functions/some examples of typical hash function techniques?

  * what are some typical operations defined on hash tables? how can a hash table be used?
  * how are items inserted into a hash table? how are items retrieved from a hash table? What is the average case time complexity for such insertion and deletion? what is the worst-case time complexity for these operations (and when does it occur)?
  * what particular typical "collection of things" operations are particularly inefficient when

hashing is used to implement that collection? [I am thinking of two in particular...]

* Be comfortable with open-addressing (plain array-based hash table) collision-resolution techniques such as linear probing, quadratic probing,double hashing, rehashing.
  * what is meant by clustering? (primary and secondary clustering)
  * why is clustering a problem?

  * does every collision result in clustering? Can one have clustering with a hash table implemented using buckets-and-chaining?

* What is the significance of hash table size in hashing's performance, in general?
  * what kind of characteristics are considered desirable for a hash table's size?

* if you decide to dynamically increase the size of the hash table in such a scheme, what must be done? (How would you do so?)

* in what kinds of situations is a hash table appropriate?
* what are some typical applications of hash tables?
* what operations in particular are NOT so well-behaved when implementing a collection of items using a hash table?

* need to be comfortable with both array-based and separate chaining/buckets-and-chaining hash table implementations.
  * you should be able to compare/contrast these implementations; discuss their tradeoffs, big (O) complexity for different operations using the different implementations, etc.

  * Given a hash function and a hash table implemented as an array of pointers to linked lists (that is, implemented using **separate chaining/buckets-and-chaining**), could you show what hashing would do for a collection of actions (insertions and deletions of specified values)? Could you do so for a hash table implemented as an array (open addressing) using a given collision strategy?

* need to be able to **use** a given hash table ADT or UML or interface to solve problems; you also need to be able to implement (and reason about the big O complexity of) hash table operations in the different hash table implementations.

* **graphs**
  * what is a graph? What is its mathematical definition? What is the relationship between graphs and trees?

  * You should be comfortable with the basic graph terminology: vertex, node, edge, graph, subgraph, adjacent, path, simple path, cycle, simple cycle, connected, disconnected, complete, undirected graph, directed graph, undirected edge, directed edge, weighted edge, successor, predecessor.
    * for the above terms for which it is appropriate, be comfortable with their (sometimes different) meanings for directed graphs and indirected graphs.

  * what are some typical operations defined on graphs? how can graphs be used?

  * in what kinds of situations is a graph appropriate?
  * what are some typical applications of graphs?

- traversals: depth-first-search (DFS)-based, breadth-first-search (BFS)-based
  - given a graph and a starting vertex, you should be able to show an order in which a graph's vertices could be visited using each of these;
  - for which is a recursive approach easier? For that traversal, if you do not use recursion, what abstract data type is particularly appropriate for use in that traversal?
  - in implementing a DFS-based traversal, which abstract data type is particularly useful/appropriate, if recursion is not used? Can recursion be used easily for this?
  - in implementing a BFS-based traversal, which abstract data type is particularly useful/appropriate, if recursion is not used? Can recursion be used easily for this?
- need to be comfortable with both adjacency-list and adjacency-matrix implementations of graphs
  - you should be able to compare/contrast these implementations; discuss their tradeoffs, big (O) complexity for different operations using the different implementations, etc.
  - you should be able to sketch a conceptual depiction of a graph within an adjacency matrix implementation or within an adjacency list implementation, whether it is directed or undirected, whether it has weighted edges or not. You should be able to perform graph manipulations to a graph expressed in such form.
    - (you should be able to read/manipulate/express a graph as a drawing or in G = (V, E) form as well)
- need to be able to **use** a given graph ADT or UML or .h file to solve problems; you also need to be able to implement (and reason about the big O complexity of) graph operations in the different graph implementations.
- if you are given a connected, undirected graph --- what is an easy way to tell if it has any cycles?
- some graph-related facts you should know:
  - if a connected, undirected graph has n vertices, at least how many edges should it have?
  - if it has exactly that many edges, what do you know that it cannot obtain?
  - if it has more than that many edges, what do you know it must contain?
- if you have a connected undirected graph with cycles and you remove edges until there are no cycles, what do you get?
- What is a circuit? What is an Euler circuit (a circuit that passes through every **edge** exactly once)? What is a Hamilton(ian) circuit (a circuit that passes through every **vertex** exactly once, except it begins and ends at the same vertex v)?

* **make** and **makefiles**
  - when/why might one choose to use a **makefile** for a C++ program? [they are more general than that, of course, but in this course we're most interested in using them in that particular area.]
  - what is the basic syntax of a **makefile**? what is a makefile **entry**?
    - what is a **target**? what are **dependencies**?
    - what character MUST precede the command(s) to be run subject to the target and

dependencies?

* in a makefile for a C++ program, what is generally the **first** target?

* what command do you use to make use of a **makefile**?
  * how does that command change depending on the actual name of the makefile?

  * what happens when you type this command?

* Given a description of a scenario, you should be able to write a basic makefile for that scenario; give a makefile, you could read it and answer questions about it.
  * this could include deducing what would be re-done if a file was changed, and then **make** was called;

* for a typical C++ program: what will be the dependencies for the corresponding executable target? what will be the command for compiling it to create that executable target?

* for a typical C++ class: what will be the dependencies for its corresponding **.o** target? what will be the command for compiling it to create that **.o** target?

* **collection implementation options**
  * at this point in the semester, you should be comfortable with a wide variety of possibilities for implementing a class for a specialized collection of items; you should be able to reason about this, discuss this, answer questions about the possibilities, tradeoffs, performance considerations, etc.

  * (you should now be prepared to make reasoned use of the C++ STL - Standard Template Library, although the STL itself will not be on the final, of course)

* **inheritance** and **derived classes**
  * the following material will **NOT** be on the final; we did not formally cover it in Week 15 this semester. HOWEVER, I am leaving it here because, if you look over this and do not recall it from CS 131, then you SHOULD re-read Chapter 14 in the course text before next Fall...

  * given code, could you identify which class(es) are derived class(es)? which are base classes?
    * given a base class and a description of a desired class that should be derived from that class, you should be able to write the declaration/implementation of that class.

  * what are the benefits of derived classes? (should know at least two... one of which particularly pertains to containers...)

  * can a class be derived from more than one other class? What is this called?

  * when is it appropriate to derive one class from another? (when they are related via an IS-A relationship...)

  * given code, would you know what members are inherited within a derived class? Could you write appropriate calls from instances of that derived class?

  * when does a derived class need to explicitly define its own copy constructor, destructor, and assignment operator?
    * when it does this, what should these explicitly call within them?

    * when these are automatically generated, what steps do these automatically-generated versions include? How does the destructor differ from the copy constructor and assignment operator in this regard?

* what is/are the type(s) of an instance of a derived class?

* should be able to write a derived class' constructor so that it explicitly/appropriately calls its base class' constructor;

* what does it mean to override/redefine an inherited function?
* what is often called --- usually as the first thing --- within the implementation of an overridden/redefined function?
* what is the syntax for allowing a derived class instance to call the base class's version of an overridden/redefined function?

* what is a virtual function? how is it indicated syntactically? what does it mean (why do it)?
    * given code involving pointers to instances of base classes and derived classes, you should be able to tell --- based on whether overridden member functions are virtual or not --- which version of an overridden member function will be called in different situations. (You'd also be able to say when late binding is occurring, and when early binding is occurring.)

* what is a pure virtual function? What is an abstract class?
    * what can you not do with an abstract class?
    * what must be done in a class derived from an abstract class, if it is not to also be an abstract class?
    * what might be some of the benefits of using an abstract class?