CIS 291 Exam #2 - Study Suggestions

* **last modified: 4-19-05**

* The exam especially focuses on HW #6 - HW #10, the Week 8 - Week 13 Lab Exercises, and material from the 3-10-05 lecture through and including the 4-14-05 lecture.

  * obviously, much of what we have done since Exam #1 builds on the material covered on Exam #1. So, some of that material will be involved in Exam #2 questions. However, the **focus** of the Exam #2 questions will generally be on the material covered since Exam #1.

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

* But, these are some especially-significant topics to help you in your studying for the exam.

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

  * there **could** be question(s) on Exam #2 like that on Exam #1 where you must give Contract, Purpose, Preconditions, Postconditions, Examples, etc.

  * that is, 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 exam will be closed-book and closed-notes, 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;

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

  * note that **demonstrations** of the various algorithms and ADT operations are definitely fair game; some examples (NOT an exhaustive list!) include showing the steps for insertion into and deletion from a heap, showing the steps for treesort and heapsort, showing what would result from computing the union of two sets (or showing what would result from calling a particular operation for a particular ADT instance), showing what binary search tree would result from a series of insertions, showing the steps in deletion from a binary search tree, showing what linked structure would result from given code, etc.

* note that you could definitely be given code and asked questions about it.

* 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++.

* some Exam #1 topics have carried through this period, and are still "current" for Exam #2;

  * for example, there could certainly again 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 a particular operation on an instance of a particular ADT containing **n** items implemented in a particular way.

  * you still should be able to read and use a data structure given a "pseudo-UML" diagram. You should be able to answer questions based on reading it, and should be able to write code using such a class (whether you know the details of its implementation or not).

  * you should be able to reason about the tradeoffs involved in different implementation possibilities for

implementing an ADT.

- * note that you are still responsible for being able to deal with **template classes** (see Exam #1 study suggestions).

* **linked lists**

  * note that this includes linked implementations of ADT's in general; for example (but not limited to) linked implementations of stacks, queues, hash tables (buckets and chaining), and trees of various sorts.

  * since linked lists depend on them, you should, of course, be comfortable with **pointers**

    * their meaning/semantics, declaration, how you set them, how you use them, how you allocate memory dynamically for something a pointer points to, how you deallocate dynamic memory that a pointer points to, etc.

    * you should be able to dynamically allocate (and deallocate) arrays as well

    * how do you use a temporary pointer to a node to "walk through" a linked list of nodes?

    * (course style standard) what should a pointer that is currently not pointing anywhere be set to?

    * what should you check about a pointer variable before you try to call the methods of the instance it (may!) point to?

    * when does a class especially need to include an explicitly-specified copy constructor, destructor, and assignment operator? Why? How?

  * be comfortable with how to set up, use a typical singly-linked list node class

  * what are the typical member variables for a singly-linked list node class? What is the head of a singly-linked list, typically? (and what is the tail of a singly-linked list?)

  * you should be comfortable with the typical conceptual drawing/picture of a linked list;

  * How do you access the member variables within a singly-linked list node? How do you set the member variables within a singly-linked list node?

  * if you have a pointer to a node, how do you access the components of the node pointed to by that pointer? How do you set the member variables of the node pointed to by that pointer?
    * what happens if you try to do the above for a NULL pointer? How should you avoid this?

  * you should be comfortable with how to perform the various typical actions discussed on linked lists (singly- and doubly-linked) (How do you "walk through" a linked list? How do you properly delete a specified node? How do you add a node? etc.)

  * what is the difference between a singly-linked list and a doubly-linked list? What are the trade-offs between them? Why might you choose one over the other (in what situations might one be preferred over the other)?

  * note that you should also be able to reason, compare/contrast, compare big-O notations, etc. about linked lists versus arrays.

  * (and, you should now be comfortable with the buckets-and-chaining (sometimes called separate chaining) implementation of hash tables, now.)

* **recursion**
    * what is a recursive definition? what is a recursive function?
    * requirements for "good" recursion! (must have at least one non-recursive "base" case; recursive cases must be guaranteed to eventually "lead" to a "base" case)
    * given a function --- does it demonstrate "good" recursion or not? Why? Why not?
    * what is a "base" case? Does every recursive function need one? Can a recursive function have more than one?
    * what is a recursive case? Does every recursive function need one? Can a recursive function have more than one?
    * given a recursive function, you should be able to tell what it would produce for a call of that function; given a specific call to that function, you can give the results of that call.
    * you may be asked to write a recursive function
* **mergesort** on **linked lists**
    * you should be comfortable with this algorithm; you should be able to describe it, reason about it, show its steps for an example list.
* **binary trees**
    * what is a (general) tree? what is a binary tree?
    * given a picture, could you tell if something is a **tree** or not? ...is a **binary tree** or not?
    * be comfortable with the basic tree-related terminology: node, edges, parent, child, sibling, root, leaf, interior node, ancestor, descendant, subtree, left child, right child, left subtree, right subtree, depth of a node, depth of a tree, level of a node, full binary tree, complete binary tree, balanced binary tree, path, length of a path;
        * for CIS 291 purposes, the root **is** indeed considered to be an interior node **IF** it is not a leaf
            * (or: the root is considered to be an interior node IF the tree has more than one node)
            * (or: the root is considered to be an interior node IF the root has at least one child)
        * I might give you a depiction of a tree or binary tree, and ask which nodes are examples of certain terms, etc.
        * why is it desirable that a tree be balanced?
    * what are some of the possibilities for how a binary tree might be implemented.
    * given a pseudo-UML diagram for a tree template class, you should be able to make appropriate use of it.
    * you should be able to describe and/or demonstrate **preorder**, **inorder**, and **postorder** traversal of binary trees.
    * what is an arithmetic **expression tree**?
        * what is in the interior nodes of such a tree?
        * what is in the leaves of such a tree?

        * if you have an **expression tree** and you traverse it in preorder, what results?
        * if you have an **expression tree** and you traverse it in inorder, what results?
        * if you have an **expression tree** and you traverse it in postorder, what results?

    * how might a binary tree be implemented using arrays? using linked nodes? Be able to compare./contrast the different possibilities.

    * some additional facts about binary trees that you should know:
        * in a balanced tree with **n** nodes, what is its depth (in rough big-O notation terms)?

        * what is the maximum depth that a tree with **n** nodes can have (in rough big-O notation terms)?

* **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?

    * What properties must a binary tree have to be a binary search tree?

    * What is the algorithm to search for an item in 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.

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

        * what sorting algorithm is based on this insight?

    * 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 diagram 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.

    * be able to describe/recognize/reason about (at a pseudocode level) the algorithm for rebalancing a binary search tree using an auxiliary array.

        * (You should know that there is also an algorithm suitable for building a balanced binary search tree from a file of sorted data, but we did not cover the details of that algorithm deeply enough for you to be held responsible for details past this.)

* **heaps**
    * what is a **heap**? What properties must a heap satisfy?

    * 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?

        * you might have to describe/answer questions about/reason about insertion and deletion from a heap; you might have to demonstrate the algorithms for heap insertion and deletion for an example heap (as you did in a lab exercise, remember).

    * 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.

    * we discussed the concept of a **priority queue**;
        * what is a priority queue? how does a priority queue differ from a "regular" queue?

        * in what kinds of situations might a priority queue be useful?

        * why might a **heap** work nicely for implementing a priority queue?

    * how is a heap used to good advantage in **heapsort**?

* **heapsort** and **treesort**

    * (note: you are responsible for non-rebalancing treesort)

    * you should be very comfortable with the pseudocode for these algorithms; you should be able to describe these algorithms (at the pseudocode level), you should be able to give pseudocode versions of these algorithms, you should be able to recognize and reason about these algorithms.

    * you should know their average and worst case time complexity, and should know how they compare to the sort algorithms discussed earlier in the semester (both in terms of space and of time).

    * you may have to show/demonstrate their steps for a specific example; you may have to reason/answer questions about those steps.

* **sets**

    * what are the distinctive characteristics of a **set** ADT? what are some of the typical operations that one would expect for a **set** ADT?

    * what are some of the options for implementing a **set** ADT? What are the tradeoffs between those options?

    * should be able to implement the typical **set** operations using a variety of different implementation

approaches;

* **internal iterators**

    * what is the difference between internal iterators and external iterators?

    * why is it useful to have internal iterator methods or external iterator classes?

    * should be comfortable with **internal iterator** methods such as those discussed in lecture on 4-14-05; (start(), is_item(), current(), advance())

    * should be able to reason about how one could implement these internal iterator methods; should be able to write such implementations (for suitable implementations of ADT's)

    * should be able to write a <u>well-structured</u> for-loop that uses all four of the internal iterator methods discussed above to traverse/"walk through" an ADT instance.

* note that you now know many more **possibilities** for how one might **implement** an ADT, given a description of its characteristics, and especially if given a **.h** file or pseudo-UML for such an ADT.

    * you should be able to reason about and/or list these implementation possibilities;

    * you should be able to reason about the tradeoffs involved in different implementations of a given ADT; these could include (but are not limited to) comparing the Big-O notations for an operation using different approaches, comparing the relative ease of implementing an operation using different approaches, comparing the space implications for different implementations.