CS 132 Exam #2 - Study Suggestions

* **last modified: 3-29-05**

* The exam covers through HW #7 (HW #4-HW #7 particularly), the Week 10 Lab Exercise (Week 6 through Week 9 Lab Exercises particularly), and material through the 3-29 lecture (2-16 lecture through 3-29 lecture particularly, including the several ways of implementing the bag class, including as a template class).

  * you are responsible for tree terminology and the basic ideas behind trees and tree traversal; you are not responsible (yet!) for implementation of trees.

  * Note, however, that if I were to give you a pseudo-UML for a binary tree template class, I could expect you to be able to use it reasonably (to write code involving declarations of instances of the template class and calls to its public member functions).

* 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 that has been covered in a **lab exercise** or **homework assignment** is **ESPECIALLY** fair game.

* 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.

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

* 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.

* You are responsible for being familiar with, and following, the course 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 a particular operation on 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.

* recursion could still come up on this exam...for example,
    * ...given a linked list of integers, how can those integers be summed recursively?

    * ...how can one recursively traverse a binary tree in preorder, postorder, and inorder? What are the base cases for such traversals?

* how should one **cast** a value to another type, using "modern" C++ type casting? (ref: Week 6 Lab Exercise)

* implementations of the **bag** class and **bag** template class
    * (be comfortable with the basic **bag** ADT and how it can be used, as well)

    * How can you use a static array to implement a bag? How can you use a dynamic array to implement a bag? How can you use a linked list to implement a bag?

    * you should be able to compare/constrast the different implementations of the bag ADT --- its static-array-based implementation, its dynamic-array-based implementation, its linked-list-based implementation (whether of a class or a template class).
        * I could, for example, ask you to give the {average-, best-, worst-)case run-time complexity for a bag of n items expressed in big-O notation.

* should be comfortable with **internal iterators** such as those discussed; should be comfortable with implementing and using the member functions related to internal iterators (and especially with using these appropriately to "walk" through a container's contents with a well-designed for-loop).
    * you are **not** responsible yet for the separate iterator class discussed in Ch. 6, however.

* **set** container class and template class

    * How can you use a static array to implement a set? How can you use a dynamic array to implement a set? How can you use a linked list to implement a set?

* you've seen several container ADT's at this point --- you should be noticing some of the items that should be included within ANY ADT, and some of the issues that should be considered in designing an ADT. There could be questions about such topics on this exam.

* There will likely be questions regarding **pointer manipulation**.
    * you should be able to dynamically allocate (and deallocate) arrays as well.

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

* **linked lists**
    * 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 memory that a pointer points to, etc.)
        * and (course style standard) what should a pointer that is currently not pointing anywhere be set to?

    * 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.)

- you should be able to do things to a linked list recursively;

- 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.

* **template functions**
* **template classes**
    * you are expected to be comfortable with these --- you may have to read and/or write them, and read and/or write code involving them.

    * why would one want to use a template function? How does a template function differ from a regular function?
        * how can template functions and template classes help to further functional abstraction?

    * what is a template parameter? what is a template prefix?
        * (note that, in lecture/lab/assignments, we've been using **typename** in the template prefix --- `template <typename ...>` That's a **course style standard**.)

    * what has been our course practice for the suffix for files containing a template function? Is there a .h file for template functions?

    * When is a template function compiled?

    * if a program is to call a template function...
        * ...what should it #include?
        * ...how does it call it?
        * ...how do you compile and link that program on cs-server?

    * why would one want to use a template class? How does a template class differ from a regular class?

    * what has been our course practice for the suffixes for the two files involved in creating a template class? What should be #include'd where?

- When is a template class compiled?

- if a program is to use a template class...
    - ...what should it #include? (And what must that #include'd file #include?)
    - ...how does it declare an instance of that class?
    - ...how does it call a public member function for an instance of that class?
    - ...how do you compile and link that program on cs-server?

- **stacks**
    - what is LIFO?

    - you need to be comfortable with the stack ADT (and its pseudo-UML).

    - what is a stack? what are the typical operations defined on stacks? how can a stack be used?

    - in what kinds of situations is a stack appropriate? what are some typical applications of stacks?

    - if I asked you to perform a sequence of pushes and pops on a stack, you should be able to simulate how it would behave and what would result;

        - note that this could be quite abstract (as done in one of the lab exercises), or it could involve showing what would happen using a particular implementation;

    - how should you avoid popping from an empty stack? what is stack underflow?

    - how can stacks be implemented? (you should know of at least 3 different ways)

    - you should be able to **use** the stack ADT in problem solutions; you also should be able to implement stack operations in the different stack implementations.

    - how can you implement stacks using static arrays? using dynamic arrays? using pointers/linked nodes?
        - you should be able to compare/contrast these implementations; discuss their tradeoffs, big-O complexity for different operations using the different implementations, etc.

    - how can stacks be used to help in various expression-based activities? in implementing recursion? in detecting palindromes? in reversing something?
        - (they can also be used to help to implement **back-tracking** in searching, although we haven't happened to get around to mentioning that yet. It is a cool use of a stack, however.)

    * note that you are **not** responsible for the C++ Standard Template Library (STL) class stack (even though I think it is included in one of the chapters assigned as a reading assignment).

* **queues**
    * what is FIFO?

    * you need to be comfortable with the queue ADT (and its pseudo-UML).

    * what is a queue? what are the typical operations defined on queues? how can a queue be used?

    * in what kinds of situations is a queue appropriate? what are some typical applications of queues?

    * if I asked you to perform a sequence of enqueues and dequeues on a queue, you should be able to simulate how it would behave and what would result;
        * note that this could be quite abstract (as done in one of the lab exercises), or it could involve showing what would happen using a particular implementation;

        * note: despite the textbook, you are expected to use the terms **enqueue** and **dequeue** for that adding and removing of items, respectively, from a queue.

    * how should you avoid dequeuing from an empty queue? what is queue underflow?

    * how can queues be implemented? (you should know of at least 3 different ways)

    * you should be able to **use** the queue ADT in problem solutions; you also should be able to implement queue operations in the different queue implementations.

    * how can you implement queues using static arrays? using dynamic arrays? using pointers/linked nodes?
        * you should be able to compare/contrast these implementations; discuss their tradeoffs, big(O) complexity for different operations using the different implementations, etc.

        * in an array-based implementation of a queue, what is rightward/downward drift? How can it be avoided? (Or: what do we mean by a **circular array**? Why is it a useful approach in implementing a queue?)

        * in each implementation, how can you distinguish between a full queue and

an empty one? [note: taking the "easy way out" with additional useful data fields is an acceptable option for this course.]

- how can queues be used to help in computer simulations? to help in recognizing certain kinds of languages? in detecting palindromes?

- note that you are **not** responsible for the C++ Standard Template Library (STL) class queue (even though I think it is included in one of the chapters assigned as a reading assignment).

* **trees** and **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;
        * I did check, and 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 important/desirable that a tree be balanced?

    * as mentioned above, you are not responsible (yet) for how a binary tree might be implemented. However, as also mentioned, if I were to give you pseudo-UML for a tree template class, you should be able to make appropriate use of it.

    * what is a binary search 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 able to describe and/or demonstrate **preorder**, **inorder**, and **postorder** traversal of binary trees.

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

* 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?