Exam I

I. Overview: Chapter 1

A. Software Quality
   1. Correctness
   2. Reliability
   3. Robustness
   4. Usability
   5. Maintainability
   6. Reusability
   7. Portability
   8. Efficiency

B. Data Structures
   1. Some algorithms require data to be in a specific structure such as a sorted list
   2. Data must be in some structure in order to be processed at all

Java and object oriented programming review and concepts

II. Universal Modeling Language (UML) (Appendix A)

A. BlueJ does this automatically to a certain extent
B. Class Diagrams
C. Relationships
D. Associations
E. Classes as aggregations of other classes
F. UML diagram for implementation of an interface
G. UML diagram for one class that uses another

III. Object Oriented Design (Appendix B)
A. Primitive data types vs Objects
   1. int
   2. float
   3. long
   4. boolean

B. Overview of Object Oriented Programming (OOP)
   1. Objects
   2. Abstraction
   3. Instantiation

C. Class libraries and packages

D. Classes
   1. public vs private
   2. scope
   3. properties
   4. constructors
   5. methods
      a. return type for a method
      b. method signature
      c. invoking (or calling) a method
      d. method (or operator) overloading

E. static Be very careful with this!

F. Wrapper classes
   1. Integer for int
   2. Allows primitives to be used as an object

G. Interfaces and the Comparable interface

H. Inheritance
   1. Used to extend an existing class
   2. May, or may not, define additional properties
   3. May, or may not, define additional methods
   4. All properties of the super class are available to the new class
   5. All methods of the super class are available to the new class

I. Polymorphism or generic programming
IV. Exceptions

A. Usually done by extending the class Exception
B. Can define and create user exceptions as needed
C. try blocks
D. catch

Analysis of Algorithms

V. Time and space complexity

A. Space (memory and/or program states) complexity is machine/OS dependent
B. Time complexity can be useful if the problem size is large, $n \to$ large.

VI. Complexity sets

A. “Little Oh”: $o(f(n))$
   1. Always larger than measured time
   2. Not very useful as $o(\infty)$ works for everything.
B. “Big Oh”: $O(f(n))$
   1. A tight upper bound
   2. Can be found by analyzing the steps required to solve the problem
   3. Means nothing for small values of $n$ and only applies as $n$ grows large
C. “Little omega”: $\omega(n)$
   1. Always smaller that measured time
   2. Not very useful as $\omega(0)$ works for everything.
D. “Big omega”: $\Omega(n)$
   1. A tight lower bound
   2. Can be found by analyzing the steps required to solve the problem
   3. Means nothing for small values of $n$
E. “Big theta”: $\Theta(n)$
   1. If a function is both $O(n)$ and $\Omega(n)$ then it is $\Theta(n)$

VII. Growth of functions
A. “Big Oh” is a set of functions, not a single function

B. If \( n \) is the base of each term and not an exponent, \( O(g(n)) \) results in polynomial time.
   1. Polynomial time example: \( f(n) \in O(g(n)) : g(n) = 23n^4 + 5n^2 + c \)
   2. Non-polynomial time example: \( f(n) \in O(g(n)) : g(n) = 5^n + c \)

**Simple Data Structures**

VIII. List data structures
   A. Unordered list
   B. Ordered List
   C. Indexed List
   D. Implementations
      1. Array
      2. ArrayList
      3. Linked List

IX. Stacks
   A. Built over an unordered list
   B. Elements are in the reverse of the order of arrival while in the stack (newest on top)
   C. Last in, first out (LIFO)
   D. Add and remove from the front (top)
   E. If you can access any element other than the top, it is not a stack
   F. If you can re-order the elements, it is not a stack
   G. Some uses
      1. Used by JAVA to keep track of the address in a program that calls a method
      2. Sneaky way to reverse a string or list by many \texttt{push(element)}
         followed by \texttt{pop()} until the stack is empty
   H. Required methods for a stack
      1. \texttt{push(element)}
a. Adds an element to the top of the stack  
b. Requires the method to have a pointer to top (or first)

2. pop()  
a. removes the element at the top of the stack and returns it to the calling method  
b. Requires the method to know and update top

I. Optional methods for a stack  
1. peek() returns the element at the top of the stack without removing it  
2. isEmpty() returns true if the stack has no elements in it  
3. size() returns the number of elements in the stack  
4. toString() could be useful while testing

J. Possible ways to implement (there are surely more)  
1. Array  
a. top starts at zero in JAVA  
b. push adds the element at top+1 and updates top  
c. pop returns the element at top and then updates top  
d. Array size+1 is the maximum number of elements in the stack  
e. size is top+1 for zero-relative  
f. isEmpty() returns -1 if stack is empty

2. Array list  
a. basically the same as array  
b. Remember: growing an ArrayList in JAVA is expensive!

3. Linked List  
a. top (acts like first) points at NULL until an element is pushed  
b. push links a new node with the data in element to the front of the list  
c. pop stores the element in first, points first at first.next  
d. Methods must update size if wanted

X. Queues  
A. Built over an unordered list  
B. Elements are in the order of arrival while in the queue (newest at tail)
C. First in, first out (FIFO)
D. Add to the tail and remove from the front (head)
E. If you can access any element other than remove from the head and add to the tail, it is not a queue
F. If you can re-order the elements, it is not a queue
G. Some uses
   1. Inter-Process Communications to keep messages in the correct order
   2. File or memory buffers are usually queues
   3. Key to any graphics algorithm that uses breadth first search (more about this when we cover trees)
   4. Lines of people, cars, kittens, etc. where you want to be fair and go by “first come, first served”
H. Required methods for a queue
   1. enqueue(element)
      a. Adds an element to the tail of the queue
      b. Requires the method to have a pointer to tail (or last)
   2. dequeue()
      a. Removes the element at the head of the queue and returns it to the calling method
      b. Requires the method to know and update head and tail
I. Optional methods for a queue
   1. peek() returns the element at the head of the queue without removing it
   2. isEmpty() returns true if the queue has no elements in it
   3. size() returns the number of elements in the queue
      a. toString() could be useful while testing
J. Possible ways to implement (there are surely more)
   1. Array
      a. Tricky and tedious to implement
      b. Difficult to keep head and tail updated properly
      c. The size of the array is the size of the queue
   2. Array list
      a. Basically the same as array
b. Remember: growing an ArrayList in JAVA is expensive!

3. Linked List
a. head and tail both point at NULL until an element is enqueued
b. enqueue links a new node with the data in element to the tail of the list and then points tail to the new node
c. dequeue stores the element in head, points first at first.next
d. Methods must update size if wanted
Exam II

Recursion and Sorting

I. Recursion

A. Characteristics of problems that have recursive solutions
B. Know at least one example of a problem with a recursive solution
C. Know at least one problem that has a better solution than the recursive solution (HINT: \( n! \))
D. General structure of a recursive solution
   1. Check for base case (stopping condition)
   2. Divide the problem into multiple sub-problems
   3. Recurse for each sub-problem
   4. Combine the sub-solutions into the final solution

II. Sorts

A. Important because many algorithms assume the input is in some particular order
B. Easy to understand and analyze algorithms
C. Insertion Sort
   1. Best case: \( O(n) \)
   2. Worst case: \( O(n^2) \)
D. Some sorts are not sensitive to the input data (selection sort, heap sort)
   1. Same time complexity for all inputs
   2. Selection sort: \( O(n^2) \)
   3. Heap sort
      a. We will talk some about heap sort week 10 if there is time
      b. \( O(n \log_2 n) \)
   4. Without any prior knowledge of the input, the best time complexity is \( O(n \log_2 n) \)
   5. Counting sort can use prior knowledge about the range of the input to achieve \( O(n) \)

III. Searching
A. Requires data to be placed in a data structure (tree, ordered list, or something else)

B. Time complexity depends upon both the data structure and the ordering of the data

Tree Structures

IV. Trees (general)

A. Definition
   1. A simple, connected, acyclic graph
   2. A simple connected graph with the fewest possible edges
   3. A simple connected graph such that if you remove any arbitrary edge the graph becomes disconnected
   4. A simple connected graph such that if you add any missing edge a cycle will result
   5. For all intents and purposes, each of the above definitions are equivalent; however, the first one is the most formal definition

B. Traversals
   1. Breadth first
   2. Depth first
      a. in-order
      b. pre-order
      c. post-order

C. Binary trees
   1. Traversals (know the pseudo code!)
      a. Breadth first
      b. Depth first
      c. in-order
      d. pre-order
      e. post-order

D. Binary Search Tree or BST
   1. Maximum depth to search is tree height
   2. Unbalanced trees: depth approaches the number of nodes
3. Balanced trees: depth approaches $\log_2(n + 1)$
   a. sorted input leads to the worst case
   b. randomizing the input may slightly improve the chances of a balanced tree, but only slightly.

E. Red-Black trees
   1. A special type of BST
   2. Effort to automatically balance the tree
      a. attempts to balance the tree on each add
      b. attempts to balance the tree on each delete
   3. Adds a “color” to each node to help detect un-balanced trees
   4. Does not guarantee a perfectly balanced tree
Exam III

I. Hashing

A. Characteristics of a good hash function
   1. Reproducible: the same input always gives the same output. This is critical, if it’s not reproducible it will not work.
   2. A unique key is needed for each piece of data, if possible.
      a. Student numbers
      b. Social Security Number
      c. Driver’s license number and state
      d. Avoid anything with today’s date as part of the key when practical.
   3. Key can be processed by a function to decrease the chances of collisions
      a. Re-arrange the data
      b. Pre-append or post-append some data
      c. Concatenate two or more unique pieces of data (such as last name and student number)
      d. A common practice was to mod a numeric key by a large prime.
      e. A common practice with alphabetic data is to use the binary equivalent as the key
   4. Quick (usually not a problem)
   5. Collisions: two different inputs producing the same output. This usually cannot be avoided completely.
   6. Produces few collisions

B. Hash code to table index
   1. Best is: index = hashcode mod tablesize
   2. Insures the index is within range
   3. Does not have to be changed when the table grows
   4. Usually produces collisions if there is enough data. Always produces collisions when the number of data entries is larger than the hash tables. (why?)

C. Hash table
   1. Up to tablesize slots usually indexed starting at zero.
2. Duplicate entries (collisions on table index) are a problem both when putting information into the hash table and when retrieving it.

3. Expensive to change the tables size
   a. Must copy every entry into the new table
   b. Must re-hash each entry
   c. Must use temporary storage equal to the size of the new table
   d. Rule of thumb: Double the tables size each time it is increased. The reasoning is beyond the scope of this class.
   e. Rule of thumb: Increase the tables size as soon as the loading reaches about 70%.
   f. Shrinking a hash table is as expensive as increasing the size. Therefore, shrinking is usually not done.

II. Hash Table Duplicates (collisions on the table index)

A. Linear probing
   1. Update:
      a. Try to place in the table at the proper index
      b. If the desired slot is occupied, continue adding 1 to the index until an empty slot is found.
      c. If there is no empty slot, grow the table and try again.
   2. Retrieval:
      a. Look in the proper table slot
      b. If the correct data (based upon the key) is not found, do a linear search

B. Overflow
   1. Update:
      a. An area is reserved after the normal table for duplicate entries.
      b. If the table slot is in use, place the data in the overflow area.
   2. Retrieval:
      a. If the information in the table slot is not the entry being searched for
      b. Linear search the overflow

C. Linked list table entries
   1. Update:
a. Try the correct table slot. If not in use, place the data in that slot with a pointer to “null”.
b. If in use, add the data to the end of a linked list which has its head as the table slot

2. Retrieval:
   a. Try the correct table slot. If it does not match...
b. Walk the linked list with the table slot entry as the start of the list

D. Double hashing
   1. Entries are hashed twice using different hash functions.
   2. If one hash value leads to a collision, the other is tried.
   3. Advantages
      a. Fewer collisions
      b. Not much slower than single hashing
   4. Disadvantages
      a. Must hash each key twice on update and retrieval.
      b. Often leads to the same overhead as multiple look-ups.
   5. You are not responsible for double hashing on this exam.

E. Other methods exist, such as quadratic probing, prime probing, and re-hash probing.
   1. Often a modification of the ones given here.
   2. Each has their advantages and disadvantages.
   3. None of these is on the exam.

III. Advantages and Disadvantages of hash tables

A. Advantages:
   1. The hash function can be chosen based upon the known characteristics of the key data.
   2. There are many well documented hash functions available.
   3. Key data can be re-arranged at will to produce fewer collisions. For example: reverse the digits of the student number if the last four are the year the student will graduate.
   4. It is usually possible to design a hash function that works in constant time.
   5. Can add or find an entry in constant time. $O(1)$. 
6. Well understood.
7. Hash table can grow.
8. Fairly reasonable use of space.
9. Can be used to implement a database.
10. If the hash function is a one-way function, the hash table can be used to securely store passwords.
   a. Access to the hashed password does not mean the password can be retrieved.
   b. Knowing the hashed value does not help to determine what password to enter to get the same hash value.
   c. Very similar as to what is used for UNIX and Linux passwords. I suspect WINDOWS uses a similar approach.

B. Disadvantages:
1. Must develop a hash function that produces few collisions
2. Must develop a hash function that always produces the same value.
3. The hash table needs to have empty (wasted) space to allow for the easy placing of new entries.
4. Growing the table is expensive in terms of time.
5. Shrinking the table is usually too expensive to be practical. (Same time complexity as growing the table.)
6. To list the data in key order requires the data to be sorted each time.

C. Indexing a hash table:
1. A list is kept of the keys and the table entry for the slot where the data is found.
2. Can be used to overcome the problems of listing the data in key order
3. An indexed hash table makes a reasonable database for small projects.
4. You can have multiple indexes into a hash table to avoid common sorts.
   a. A student data base might use a hash table with a hash function based upon student number.
   b. An index of student numbers vs table index would allow for quickly listing the student in numeric order in $O(n)$.
   c. An index of student names verses table index would allow look up by name or alphabetic listings in $O(n)$.

D. Java collections provide multiple hash tables.
IV. Sets

A. A collection of unique things (elements). In Java, primitives or objects.

B. There is no requirement that the elements be comparable.

C. Duplicate elements are treated as one element. If the elements are keys to other data, this means that one or the other set of data will be lost.

D. Important special sets (not on exam)
   1. There is only one set with no elements and it is called the empty set: \( \emptyset \) or \{\}.
   2. The set of all possible elements is called the universe and is sometimes written as \( U \).

E. Can be stored in many different ways depending upon future use.
   1. Set (in Java collections)
   2. Ordered list
   3. Unordered list
   4. Indexed list
   5. Array (or ArrayList)
   6. Tree
   7. Heap
   8. Hash table
   9. or you can get creative.

F. Java collections provide multiple ways to deal with sets.

G. Uses of sets in programming
   1. Graphs are sets of vertices and edges
   2. Lists are sets
   3. Trees are sets because they are graphs
   4. A set can have zero, one, or many elements, so every variable or object can be a set.

H. Programming issues with sets (difficult and/or tedious)
   1. Merging sets: \( C = A \cup B \) must watch out for duplicate elements
   2. Finding an element in a set: requires a linear or binary search
   3. Set equality has no good algorithm if the sets are not ordered (\( O(n^2) \))

I. Basic operations on sets (not on exam)
1. Cardinality (number of elements in a set) is usually written: $|A|$.  
   a. $|\emptyset| = 0$  
   b. $|U| = \infty$.  
   c. Careful, sometimes the universe is limited by the context of the question. For example: The universe of “K” students is not infinite.

2. Union $A \cup B$  
   a. all elements that are in either $A$ or $B$  
   b. $x \in (A \cup B) \rightarrow ((x \in A) \lor (x \in B))$  
   c. $|A \cup B| \leq |A| + |B|$

3. Intersection $A \cap B$  
   a. all elements that are in both $A$ and $B$  
   b. $x \in (A \cap B) \rightarrow ((x \in A) \land (x \in B))$  
   c. $(|A \cap B| \leq |A|) \land (|A \cap B| \leq |B|)$

4. Compliment $\overline{A}$ or $A^c$ (everything not in set A).

5. Set equality. $(A = B \equiv (A \subseteq B) \land (B \subseteq A))$  
   a. Can be done in $O(n)$ if the elements are comparable and stored in an ordered fashion.  
   b. Can be done in $O(n^2)$ if the elements are not comparable or stored in an unordered fashion.

6. Special properties of sets  
   a. $(\overline{A}) = A$  
   b. $A \cup \overline{A} = U$  
   c. $A \cap \overline{A} = \emptyset$  
   d. $A \cup U = U$  
   e. $A \cap U = A$  
   f. $A \cup \emptyset = A$  
   g. $A \cap \emptyset = \emptyset$

V. Heaps and HeapSorts  

A. Characteristics of a heap  
   1. Binary tree  
   2. Each node is the root of a heap  
   3. Each node is greater (smaller) than its children nodes

B. Heaps stored in arrays
C. Building the heap
D. Re-establishing the heap
E. Heap Sort
   1. $O(n \log_2 n)$
   2. Not stable
   3. No best/worst case
F. Other uses of a heap
   1. Priority queue
   2. Well-ordered sets

VI. Anything else that came up in:
A. Car Wash Lab
B. N Queens Lab
C. BLAST Lab
D. Any MiniLab due after Exam II.
E. Chapter 14 from Edition 3
F. Lectures on sets (Editions 3 and 4 treat sets in different places so there was no reading assigned.)