COMP210 Review Topics for Spring 2018

Exam I

I. JAVA Review

A. Data Types
   1. primitives
      a. int
      b. boolean
      c. float
      d. long
      e. others
   2. Objects predefined in JAVA
      a. Inherit from Object
      b. String
      c. Integer
      d. others
   3. User-defined
      a. Defined in class files
      b. Inherit from Object
      c. Do not exist until you instantiate them
   4. public, private, and protected
   5. static
      a. Avoid using static
      b. Only use for things like assigning serial numbers

B. Casting
   1. Forces an object to a different object type
   2. Comparable
   3. Object

C. Appendix B in the book
D. Try-catch
E. UML diagrams
F. Object Oriented Programming
   1. User-defined reusable objects
      a. Defined in class files
      b. Inherit from Object
      c. Do not exist until you instantiate them
      d. properties (data items)
      e. methods
      f. method signatures
      g. each instantiated copy gets its own properties and methods
      h. method and operator overloading
   2. Scope
      a. Global variables
      b. Avoid global scope variables
      c. Sometimes useful for constants to have global scope, e.g. \( \pi = 3.14159 \)
      d. Local variables

G. Generic programming

II. Complexity
   A. Almost always a function of the number of data points \( n \)
   B. As \( n \) grows large
   C. Cases
      1. Best case
      2. Mean case (usually random input)
      3. Worst case
      4. Almost always interested in worst case
   D. Space complexity
   E. Time complexity
      1. Growth of Functions
      2. Quick and dirty methods to determine
      3. Big “O” and friends
         a. \( o(f(n)) \) Upper-bound on growth
b. $O(f(n))$ Tight upper-bound on growth

c. $\omega(f(n))$ Lower-bound on growth

d. $\Omega(f(n))$ Tight lower-bound on growth

e. $\Theta(f(n))$ Tight upper and lower bounds on growth

4. Most of the class will be spent in finding data structures and algorithms to improve worst case behavior

III. Lists as the most basic data structures

A. Unordered lists

1. Basis for other data structures
   a. Queue
   b. Stack
   c. Set
   d. Tree
   e. others

2. Not set in ascending/descending order by some key data

B. Ordered lists

1. Basis for other data structures

2. Many algorithms require input data to be in a specific order

3. Must be built in order or sorted

C. Indexed lists

1. May be accessed by position in the list (arrays, array lists, etc.)

2. May be ordered or unordered

D. Linked lists

1. Many only be accessed sequentially from the first element

2. Grow and shrink efficiently

3. Underlying structure for other data structures
   a. Queues
   b. Stacks
   c. Trees
   d. Sets

4. Must keep track of the “first” or “head” of the list

5. Might be useful to keep track of the “tail” or “last” of the list
6. Singly linked lists may be traversed from the “first” to next
7. Doubly linked lists have links to next and previous items in the list

IV. Stacks and Queues

A. May be built over any unordered list
   1. Arrays
   2. Linked Lists

B. Stack
   1. Last in, First out (LIFO)
   2. Easy to implement over linked lists
   3. Easy to implement over arrays
   4. Required methods
      a. push(element) places an element on the top of the stack
      b. pop() removes and returns the top element
   5. Optional methods
      a. peek() Returns the top element in the stack but does not remove it
      b. isEmpty() Returns “true” if there are no elements in the stack
      c. size returns the number of elements in the stack

C. Queue
   1. First in, First out (FIFO)
   2. Easy to implement over linked lists
   3. Can be difficult to implement over arrays
   4. Required methods
      a. enqueue(element) places an element at the tail of the queue
      b. dequeue() removes and returns the first element in the queue
   5. Optional methods
      a. peek() Returns the first element in the queue but does not remove it
      b. isEmpty() Returns “true” if there are no elements in the queue
      c. size returns the number of elements in the queue
V. 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

VI. 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 \( O(cn \log_2 n) \)
   5. Counting sort can use prior knowledge about the range of the input to achieve \( O(n) \)

VII. 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

VIII. Trees (general)

A. 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 (all three, see above)

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 $\log_2(n + 1)$

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

IX. 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: $\text{index} = \text{hashcode} \mod \text{tablesize}$
   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 table size. **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 table 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 table size each time it is increased. The reasoning is beyond the scope of this class.
      e. Rule of thumb: Increase the table 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.

X. 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.

XI. 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. Use them!
   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 \textit{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.

XII. 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 up future use.
   1. **Possibly on the exam.**
   2. Ordered list
   3. Unordered list
   4. Indexed list
   5. Array (or ArrayList)
   6. Queue
   7. Stack
   8. Tree
   9. Heap
   10. Hash table
   11. Variable (if only one element)
   12. or you can get creative.

F. Venn Diagrams

G. 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. \(A \cap \emptyset = \emptyset\)

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

I. Uses of sets in programming.
   1. **Possibly on the exam.**
   2. Graphs are sets of vertices and edges
   3. Lists are sets
   4. Trees are sets because they are graphs
   5. A set can have zero, one, or many elements, so every variable can be a set.

XIII. Heaps and Heap Sorts **Not on the exam**

   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

XIV. 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. Lecture on sets (Editions 3 and 4 treat sets in different places so there was no reading assigned.)