CS162 Study guide for Quiz 1-5

Quiz 1 will include the topics (covered in Week 1 - Week 5) provided as below:

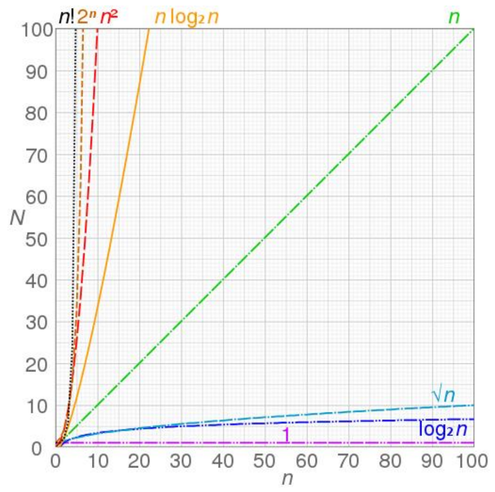
1. Big-O notation and amortized analysis
2. ADTs - Stack, Bag, Queue, Deque, Iterators
    - Description of behaviors, interface, implementation, the complexity of operations, and comparing
      implementations of ADTs with data structures.
3. Data Structures - Array, Dynamic Array, Linked List
    - Implementation of ADTs using Dynamic Array, Linked List, and Ordered Array
    - Variations of Linked List (Singly, Doubly, Circular)
    - Non-sentinel versus Sentinel
    - Compare the complexity of operations among implementations
4. Binary Search

Module 1: INTRO TO DATA STRUCTURES MASTERED

1. Describe the general content and purpose of this course. Why are data structures important?
    a. Data structures is one of the most important courses we will take as an undergrad if we plan to move on to do any kind of programming in our lives.
    b. Because the importance of well-organized data and lack of understanding of data structures when learning to program, this course allows us to move from being a hobbyist programmer to a budding professional.
    c. We will become familiar with a collection of foundational data structures.
    d. We will understand how to analyze and manage the complexity associated with data structures and their operations.
2. Describe what data structures are:
    a. What are some real-life analogs to data structures in computer science?
        i. Stacks are like mail/paper being stacked on top of each other or pancakes. Queues are people in line at a grocery store.
        ii. These are linear storage structures. Stuff is organized one after the other and we can use different techniques for accessing it, like looking at the newest, oldest, or sticking labels on it and sorting it.
        iii. 20 questions turns out to be a giant decision tree
    b. What are some data structures you might have already used?
        i. Lists, graphs, arrays, dictionaries.

Module 2: COMPLEXITY ANALYSIS MASTERED

1. Describe what time complexity is.
    a. Being able to compare data structures in a way that allows us to make an informed choice between the different data structures we might want to use for problems.
2. What is Big O notation?
    a. Big O notation is used for complexity analysis to assess a data structure or algorithm's performance.
    b. It is a tool for characterizing a function in terms of its growth rate.
    c. Think "upper bound" as the known growth order of the function, typically specified in a mathematical function.
    d. A function g(x) provides an upper bound/specifies a growth order for another function f(x) if there is some value x_0 beyond which g(x) > f(x) for ever x>= x_0.
3. Do coefficients matter in Big O notation?
    a. No, there is no sufficiently large constant such that as N gets large enough, constants and low-order terms do not matter.
4. Evaluate what are acceptable upper bounds for algorithms.
    a. Given a very large data set, what is the first time complexity at which you would say this operation isn't reasonable?
        i. Quadratic complexity. Larger sets become impossible over time.
        ii. Otherwise nlogn possibly.
    b. How much of an impact does processor speed have on time algorithm completion time?
        i. Highly. The speed at which a processor will complete/execute a calculation will sum together. If a processor takes a couple of milliseconds for each calculation, the sum of the time for each calculation will result in the overall execution time.



Module 3: LEVELS OF ABSTRACTION MASTERED

1. What are the operations of a dynamic array?
    a. Get: gets the value of the element stored at the given index in the array
    b. Set: sets/updates the value of the element stored at a given index in the array
    c. Insert: inserts a new value into the array a given index
        i. Different from set in that no existing array values are overwritten by insert. Instead, insert moves all subsequent elements down one spot in the array to make room for the inserted one.
        ii. Sometimes dynamic array implementations limit insertion to a specific location in the array (like only at the end).
    d. Remove: Removes an element at a given index from the array and moves all subsequent elements up one spot in the array to fill in the "hole" left by the removed one.
        i. Sometimes, dynamic array implementations avoid moving elements up a spot by only allowing the last element to be removed.
            1. Makes the array more limited in its use but improves its performance.
        ii. Swapping elements to fill holes instead of shifting all elements is possible. Works in a bag ADT, breaks any sort of order that might have existed.
2. What is the main difference between a static array and a dynamic array?
    a. Static arrays cannot change their capacity even after the size == capacity.
    b. Dynamic arrays will change their capacity to accommodate for a large size input. The change is dependent on the implementation, but we often consider it as a doubling of the capacity for each time the size == capacity.
3. Is Python's list datatype implementation a dynamic array?
    a. Yes! It uses a dynamic array to implement the list.
4. What is amortized analysis?
    a. Amortized analysis is the method to analyze an operations' complexity or how much of a resource (especially time or memory) it takes to execute. This is done by computing the average cost over the entire sequence.
    b. Cost per operation must be known, and it must vary in which many of the operations in the sequence contribute little cost and only a few operations contribute a high cost to the overall time.
5. How can you calculate the amortized runtime for an insert operation of the dynamic array?
    a. Cost for an insert operation of a DA is incurred in two places:
        i. Inserting a new value into the DA when it already has enough capacity to hold the new value.
        ii. Copying values from old storage to the new storage during resizing.
    b. Thus, knowing the write cost and copy cost, tabulating the cost is the summation of the cost from all those operations and will result in the average amortized runtime for an insert operation of a DA.
    c. Over n total insert operations, the total cumulative cost of initial writes will be n while the doubling of the capacity of the data storage array each time we resize before we can hold n elements is logn because n=2 → log2 = 1, n=4→log4 = 2, n=8→ log8 =3
    d. However, we know that we aren't just copying a single element with each of these calls, and need to copy several elements from the existing array. Thus we deduce the k'th

copy method call incurs a copy cost of 2^(k-1). The kth cost is also equal to n/2 because the array capacity will double to achieve a size of n and there would be half that number of existing elements to copy to the new array. Therefore, k = logn.

   e. We can deduce that the sum of the geometric series is n-1. So the overall cost of the entire sequence of n insert operations—equal to the total, the cumulative cost of initial writes plus the total cumulative copy cost is: **2n-1**
      i. This represents the total cost over the entire sequence of n operations. Computing the amortized cost as the average of this total cost over n operations is:
         1. (2n-1)/n = 2n/n -1/n = 2-1/n < 2 → O(1) amortized runtime complexity.

6. Describe what an ADT is.
   a. ADT stands for abstract data type. It is an object whose behavior is defined by a set of value and set of operations. Simply, it is a representation of some data and operations on that data.
7. What are some examples of ADTs?
   a. Lists, stacks, sets, bags, static/dynamic arrays.
8. Do ADTs tell us anything about how long operations take or how much memory is used?
   a. It does not specify how data will be organized in memory and what algorithm will be used for implementing the operations.
9. Describe and implement the Bag ADT.
   a. A bag ADT is a non-organized collection of data that is analogous to a physical bag of stuff.
10. What operations does the Bag ADT support?
    a. Add(item): this takes the supplied item and adds it to the contents of the bag.
    b. Remove(item): This looks for the specified item and removes one copy of it from the bag.
    c. Clear(): removes everything from the bag. Basically, turn it upside down and shake out the contents.
    d. Count(item): counts the number of a particular item in the bag.
    e. Size(): gets the total number of items in a bag.
    f. Display(): prints all the contents of the bag.
    g. Equal(second_bag): compares the content of the current bag with the one provided by the user.
11. How can a list data structure be used to implement a bag?
    a. Lists are simply dynamic arrays. Storing data in lists is just organized. In a bag you can ignore the organization of the data in the structure. The feature operations in a bag adt can mimic a list by treating the list as a dynamic array.

Module 4: DEQUES, STACKS, QUEUES, LINKED LISTS MASTERED

1. Describe linked lists and evaluate where it is appropriate to use them.
    a. Linked lists are linear data structures, where data forms a linear sequence with individual data elements placed one after the other within the data structure.
    b. They are comprised of operations such as:
        i. Get by value: Inspecting each node starting from the first node and stopping at the node which contains the value.
        ii. Get by index: Inspecting each node starting from the first node and stopping at the node related to the index.
        iii. Add: adding a link after finding the link you want to insert after.
        iv. Remove: removing a link and linking the previous node to the next next link
    c. They are advantageous for functions where constant time is necessary. Because you can always add and remove from the start of a linked list in constant time, there is no need to resize the structure and waste time copying like in dynamic arrays.
    d. The largest drawback is needing to access elements in the middle of the list. Lists are discontiguous in memory, so you can only access each element by traversing through each node one by one to find an elements location.
    e. Additionally, a bit of space is needed for each link, which can potentially double the amount of memory used.
2. What sorts of tasks are well suited to linked lists?
    a. It is best to use linked lists where you want to use a stack or queue where operations are done only on the ends of the lists, and it is okay to use a little extra memory to hold the data.
3. Where might we want to use an array over a linked list?
    a. When you want to want random access to elements, when you know how many elements will be added/evaluated.
4. Explain the operations that a queue, stack, and deque allow.
    a. Stack
        i. Last in first out order on elements. Think pancakes or a stack or paper.
        ii. Operations:
            1. Push(x): pushes an item onto the top of the stack.
            2. Pop(): removes the top item from the stack and returns its value.
            3. Peek(): returns the value of the top item in a stack without removing it.
    b. Queue
        i. First in first out order on elements. Think a line at a grocery store for check out.
        ii. Operations:
            1. Enqueue(x): adds x to the back of the queue.
            2. Dequeue(): removes the first item in the queue and returns its value.
    c. Deques
        i. Four operations to remove/add at the front/back.
5. What are the differences between a queue and a stack?
    a. Queue's first in first out
    b. Stack's last in first out
6. How can we access something in the middle of a stack?

    a. Having two stacks and moving everything over to the other one, access the value in question, and then move all the items back. Otherwise, pop all the values until you get it.

7. Implement a linked list with a queue, stack, or deque ADT.
8. What are the steps required to delete an item at the beginning of a linked list?
    a. Set the self.head.next to self.head.next.next
    b. In other words:
        i. Define the node you want to remove (the self.head)
        ii. Set the head as the next node
            1. Self.head = self.head.next
        iii. Then remove the node by setting it to None.

Module 5: ITERATORS AND BINARY SEARCH MASTERED

1. Describe the functions that an iterator provides.
    a. Iterators is a data type that acts as a companion to a collection.
    b. It provides a mechanism to iterate through the given collection.
    c. It is implemented in a way to have access to the internals of the collection, allowing it to iterate through it.
    d. Every kind of collection will have its own iterator data type.
2. Given an iterator pointing to the 5th element in an ordered collection, which elements can we access?
    a. Iterators pick up from where you leave off. If you start at the 5th element, you continue from 6th to nth elements, disregarding the elements prior to the 5th element.
3. Evaluate when it is appropriate to use an iterator.
    a. It is appropriate to use an iterator when needing access to elements in a collection.
4. What advantages does using an iterator provide over directly accessing a data structure?
    a. Iterators allow you to pick up from where you left off and traverse on, guaranteeing that you have accessed each element within the collection from your starting point onward.
5. In what situations will an iterator not be helpful when working with a data structure?
    a. When you desire accessing an element you have already accessed.
    b. When you want to guarantee the order at which you visit elements in a collection.
    c. When you want to skip certain elements.
6. Explain the steps involved in a binary search.
    a. Compare the value to the value at the exact midpoint of the array.
    b. If it matches, O(1), perfect!
    c. Otherwise,
        i. Compare if the value is greater than or equal to the midpoint value,
            1. If greater, repeat using only the upper half of the array.
            2. If less, repeat using only the lower half of the array.
            3. If the array under consideration has size 0, stop and state the value DNE.
7. In a binary search, which elements would take the longest to find?
    a. Elements at the very front and the very end.
8. Why can't we use a binary search on an unordered collection?
    a. Binary search depends on comparing the value of interest to a specific element. Then it splits the collection to traverse the half that is either greater or less than the value of interest. Unordered collections cause this to fail because comparisons are impossible.
9. Why doesn't it make sense to use a binary search with a linked list?
    a. Binary search requires random access to elements to quickly access the middle element. Linked lists require traversal from the head on. This would take too long for large values of n because you need to travel n/2 to reach the middle, compare, and then split, then travel n/4, n/8 etc.