CS 310: Midterm Review

Connor Baker

February 27, 2019

Contents

1 Reading List

- Big-O
  - Chapter 5: Algorithm Analysis.
  - Section 5.3: You are not required to be able to prove the theorems, but you need to understand them.
- Lists
  - Section 6.1-6.3, 6.5: General Lists.
  - Chapter 15: Dynamic Array / Array List and Iterator.
  - Chapter 17: Linked List.
  - Section 6.6: Stack/Queue Concepts.
  - Chapter 16.1-16.3: Stack/Queue Implementation.
- Trees:
  - Section 18.1: Tree Basics, Binary trees.
  - Section 7.1-7.3: Recursion.
  - Section 18.3-18.4: Tree Recursions, Tree Traversals.
- Hashing:
  - Section 20.1-2: Hashing Basics and Hash Functions.
  - Section 20.5: Separate Chaining.

2 Sample Questions

2.1 Big-O

1. What are the two types of complexity we discussed in class for assessing algorithms? Give an example of trading off one for the other using a data structure discussed in CS310.
2. Explain time/space complexity using a real life example (like the time complexity examples we did in class).
3. Convert the following functions to Big-O:
   (a) $10n + O(\log(n))$
   (b) $0.25n! + 2n$
   (c) $1 + 2 + 3$
   (d) $5\log(n)$
   (e) $\log(n^2) + 100$
4. For all the code you write for other problems in this review, compute
   (a) the function which describes the time your function will take to run;
   (b) the Big-O of the function.

   Make sure to state what your variables mean (e.g. "what is n?").
5. If I stated that the Big-O of a method was $O(1)$ and that it also contained a loop, describe that loop.
6. What's the difference between "best case", "worst case", and "average case" Big-O?
7. Explain the formal definition of Big-O: $T(n) \in O(F(n))$ if there are positive constants c and n_0 such that, when $n \geq n_0, T(n) \leq cF(n)$.

2.2 Lists

8. Explain the difference between a dynamic array list and a linked list.
9. Compare and contrast singly and doubly linked lists, including the pros and cons of using them.
10. Explain the difference between a "node" class and a "linked list" class. Include in your description the terms: node, next, value, head, and tail.
11. Explain what the memory looks like for a list stored as a [static array | dynamic array | linked list].

    Make sure to represent the memory needed to store additional helper variables (such as size, head, etc.).
12. Write the code to perform [`set()` | `get()` | `add()` | `append()` | `insert()` | `remove()` | `contains()` | `indexof()`] for a [dynamic array list | singly linked list | doubly linked list].
13. Explain why we say that the worst case Big-O of appending to a dynamic array list is $O(n)$ while its amortized cost is only $O(1)$.
14. Given an array representing a list, write the code to convert it into the equivalent linked list.
15. Write the code to perform an [insertion | selection] sort on a singly linked list (with optimal Big-O).
16. Describe how you make a class "iterable" in Java.
    (a) What interfaces do you need? What methods are required by those interfaces?
    (b) Are there any additional optional methods?
17. Write code to implement a basic iterator for a(n) [array | linked list] and show how the iterator would be used.

2.3 Queues and Stacks

18. Explain the difference between a queue and a stack.
19. If we add `'a'`, `'b'`, `'c'`, `'d'` onto a stack and invoke `pop()`, what will be returned?
20. If we add `'a'`, `'b'`, `'c'`, `'d'` into a queue and invoke `dequeue()`, what will be returned?
21. Explain how to implement a [queue | stack] with a [singly | doubly] linked list with the optimal Big-O? Which end(s) should you insert/remove from for the optimal Big-O?
22. Explain how to store a stack in an array with the optimal Big-O.
23. Explain how a circular queue works for storing a queue in an array.
24. Given a series of adds and removes into a [stack | queue] stored in a(n) [array | singly linked list | doubly linked list], draw a representation of the resulting memory.

    Remember to include any additional storage you need for helper variables such as size, top, etc.
25. Write the code to perform [add | remove | isEmpty | peek] for a [queue | stack].

2.4 Trees

26. Explain the following terms: trees, root, descendants, ancestors, leaves, siblings, parent, child, depth, node height, tree height, full tree, (nearly) complete tree, perfect tree, balanced tree.
27. Given a tree, identify the [root | leaves | tree height | tree size]. Given a tree, label each node with its [depth | height]. Given a tree, identify if it is [full | (nearly) complete | perfect | balanced].
28. Explain how to store a binary tree using an [array | linked structure].
    (a) For an array, explain how to find the child/parent of a given node based only on the index of that element.
    (b) For a linked structure, write the code for a node.

    Can you generalize the idea to k-ary trees?
29. Write a recursive method to calculate the [size | height] of a tree.
30. Given a tree, perform a [pre-order | post-order | in-order | level-order] walk of the tree, printing the nodes.
31. Write code for a recursive [pre-order | post-order | in-order] tree traversal of a binary tree that prints the node values out. Assume a generic `Node<T>` class with a data field and a array of node children.

2.5 Hashing

32. What is the goal of hashing? (i.e. Why would we want to hash something into a hash table?)
33. Explain the difference between a hash code, a hash function, and a hash table. How does a hash code differ from the index of a hash table?
34. Create a hash function for [strings | numbers | cards | people] which generates relatively unique values.
35. Explain Java's hash-contract and why it is necessary.
36. Explain the idea of "load" on a table and how it relates to the big-O of the hash table's [`add()` — `remove()` | `contains()`] methods.
37. Hash [strings | numbers | cards | people] into a hash table of size [some number] using separate chaining and the hash function you wrote above.
38. Rehash your tables above to [a prime number over double the size].
39. Write code for an [`add()` | `remove()` | `contains()` | `rehash()` | `getLoad()`] method in a hash table which uses separate chaining.

2.6 General Programming and Java Specific

40. In Java, what is the Big-O of concatenating two string with + operator? Assume n is the length of the first string and m is the length of the second string.

41. Explain how to define a generic class in Java and why it is useful for data structures.

42. Explain recursion to someone who has never heard of it before. Make sure your answer includes the following terms: base case, recursive case, readability, efficiency, and iteration.

43. Given an iterative function, write the same function using recursion. Given a recursive function, write the same function using iteration.

44. Explain how you can walk backward in a singly linked list using either recursion or a stack. Why does recursion work without needing to make your own stack?

3 Sample Answers

3.1 Big-O

1. We discussed both time and space complexity (we also discussed implementation complexity).

   There is an inherent tradeoff between the speed with which an algorithm operates and its memory usage. As an example, a dequeue allows for $O(1)$ insertion at both ends of the structure, but it is typically implemented using a linked list which has a large memory footprint.

2. Suppose you're baking a cake, but you have a tiny kitchen. It would be faster to put all the ingredients you need out on the counter and prepare everything at once, but you don't have the counter-space to do it.

   As a result, you're forced to go to the fridge every single time you need something – the available counter-space (memory) directly affects how many time-wasting trips you'll need to make to the fridge (which is analogous to wasting CPU cycles because the processor has to fetch or calculate something before it can resume the algorithm).

3. Convert the following functions to Big-O:

   (a) $O(n)$ since $O(n) > O(\log(n))$

   (b) $O(n!)$ since $O(n!) > O(n)$

   (c) $O(c)$ since $aO(c) \in O(c)$, when $a \in \mathbb{R}$

   (d) $O(\log(n))$ since $a \cdot O(\log(n)) \in O(\log(n))$, when $a \in \mathbb{R}$

   (e) $O(\log(n))$ since $O(\log(n^2)) + O(100) = O(2\log(n)) + O(c) \in O(\log(n))$

4. For all the code you write for other problems in this review, compute

   (a) the function which describes the time your function will take to run;

   (b) the Big-O of the function.

   Make sure to state what your variables mean (e.g. "what is n?").

5. The number of iterations of the loop is invariant with respect to the surrounding method; that is to say that it executes a fixed number of times.

6. A best case scenario is one in which the state of an object is such that it causes the method to execute as quickly as possible. An example would be finding the element you were looking for at the head of a linked list, which means that you avoid traversing the entire list.

   A worst case scenario is the opposite of a best case scenario. An example would be finding the element you were looking for at the end of a linked list, having made you traverse the entirety of the list.

   An average case uses probabilistic analysis to determine, given any reasonable input, what the typical runtime is.

7. Suppose we have two functions, f and g, which operate on some domain $\mathbb{D}$, and let $x \in \mathbb{D}$. Then, if there there exists some constants c and $y \in \mathbb{D}$ such that for all $x \geq y$, $g(x) \leq c \cdot f(x)$, we say that $g(x) \in O(f(x))$. This can be interpreted as the fact that there is a scalar multiple of f (that'd be $c \cdot f(x)$) which is consistently larger (and therefore grows more quickly than) g for all elements of the domain past some point (which is y).

3.2 Lists

8. All `C`-style arrays are pre-allocated blocks of contiguous memory. A dynamic array is dynamic only in the sense that the memory was allocated at run-time, and that the array can "grow" (which amounts to allocating a larger block of memory and copying the contents of the previous block over to it).

   A linked list consists of elements which store references to their successors (and predecessors, in the case of a doubly-linked list). Their size is flexible because each element is created on demand. Unlike `C`-style arrays, the memory occupied by the data structure is not contiguous.

9. Singly-linked lists use less memory than doubly-linked lists because they do not store a reference to their predecessor. However, unlike doubly-linked lists, singly-linked lists are unable to traverse backwards through themselves because each element lacks a reference to their predecessor.

   Doubly-linked lists could retrieve the element before the tail in $O(1)$ time, while a singly-linked list would take $O(n)$ time to do it. This is an excellent example of time/storage complexity tradeoffs.

   For certain applications, it is desireable that an iterator be able to traverse the list in both directions – in this case, a doubly-linked list should be used.

   For other applications, memory usage is more important, which pushes a solution towards a singly-linked list.

10. A node class defines what the elements of a linked list might look like; the kind of data (value is not as generic – we could store an array in a node, which is a kind of data, but not a singular value) they might hold, whether they have a link to their predecessor, successor, or both.

    A linked list is built from a series of linked nodes. A linked list class typically consists of a reference to the head of the list (the only node without a predecessor) and the tail of the list (the only node without a successor) – having access to both the head and the tail ensures $O(1)$ time removal and insertion at either end.

11. A static array is an array which has memory that is allocated when the program is loaded into memory – its size was known at compile time. The memory used can be thought of as a contiguous block.

    A dynamic array is an array which has memory allocated during run-time: the size is not known at compile-time because it depends on user input, computations done by the program, or for any of a different number of reasons. Dynamic arrays are created by using the `new` keyword. Like static arrays, they are also contiguous blocks of memory.

    A linked list, by comparison, has a much more complex memory layout. Because each node is allocated at run-time, the memory occupied is not contiguous. The linked list object will typically consist of a block of memory which contains all of the methods of the class, as well as a variable holding the size, and a reference to the head and the tail of the linked list. The actual contents of the linked list are scattered throughout system memory (more accurately, wherever the `JVM` found space to store them).

12. Write the code to perform [`set()` | `get()` | `add()` | `append()` | `insert()` | `remove()` | `contains()` | `indexof()`] for a [dynamic array list | singly linked list | doubly linked list].

    Waiting on hearing if sharing these would be an honor code violation.

13. Big-O models the time it takes an algorithm to finish running in its worst case scenario. With respect to an `ArrayList`, the worst case would involve running out of space in the currently allocated block of memory, in which case the object allocates somewhere in the neighborhood of double the memory and copies over the data stored in the old block of memory (which takes $O(n)$ time). Usually there's more than enough room and resizing happens very rarely. As such, the amortized cost is only $O(1)$ (that is, the *expected* cost of each operation – this is *not* the same thing as the average cost).

14. Given an array representing a list, write the code to convert it into the equivalent linked list.

    Waiting on hearing if sharing these would be an honor code violation.

15. Write the code to perform an [insertion | selection] sort on a singly linked list (with optimal Big-O).

    Waiting on hearing if sharing these would be an honor code violation.

16. Describe how you make a class "iterable" in Java.

    This depends on what you mean by iterable.

    If you mean a class that can be iterated over, then you need only need to add `implements Iterator<E>` to the class and implement `hasNext()` and `next()`. (Both `remove()` and `forEachRemaining()` have default implementations and don't need to be implemented.)

If by iterable you mean that the class `implements Iterable<T>`, then all that is needed is to create a method `Iterator<T> iterator()` which returns an iterator for the object is was invoked on. Of course, if you don't have an iterator, you'll need to build one, either with an anonymous class or in some other fashion.

17. Write code to implement a basic iterator for a(n) [array | linked list] and show how the iterator would be used.

    Waiting on hearing if sharing these would be an honor code violation.

3.3 Queues and Stacks

18. Explain the difference between a queue and a stack.

    Queues are FIFO (first in, first out) data structures. Elements are added at the tail and are removed from the head. An example would be people waiting in line at a carwash.

    Stacks are LIFO (last in, first out) data structures. Elements are added and removed from the tail. Think of it like a tower of plates – plates are put on the top and removed from the top. (This will only make sense to you if you're not the special kind of maniac that tries to grab plates from the bottom.)

19. We would get the character `'d'`.

20. We would get the character `'a'`.

21. Implementing a queue with a linked list (singly or doubly doesn't particularly matter – the only difference is ensuring that nodes recognize their predecessor as well as their successor):

    - Add elements only to the tail of the linked list
    - Remove elements only from the head of the linked list

    for optimal Big-O items should be removed or inserted at the head (guarantees $O(1)$ time).

    Implementing a stack with a linked list (singly or doubly doesn't particularly matter – the only difference is ensuring that nodes recognize their predecessor as well as their successor):

    - Add elements only to the tail of the linked list
    - Remove elements only from the tail of the linked list

    for optimal Big-O items should be removed or inserted at the tail (guarantees $O(1)$ time).

22. Add or remove items only at the after index of the previously added or removed element.

    This avoids the problem of shifting all the elements upon adding or removing (which would happen if we allowed adding or removing at the front).

    This also allows for an adding and removing to take $O(1)$ time (amortized of course, since the worst case is $O(n)$ as we'd need to resize the array).

23. Explain how a circular queue works for storing a queue in an array.

    Revisiting later.

24. Given a series of adds and removes into a [stack | queue] stored in a(n) [array | singly linked list | doubly linked list], draw a representation of the resulting memory.

    Remember to include any additional storage you need for helper variables such as size, top, etc.

    Revisiting later.

25. Write the code to perform [add | remove | isEmpty | peek] for a [queue | stack].

    Waiting on hearing if sharing these would be an honor code violation.

3.4 Trees

26. Explain the following terms:

- Tree: A set of nodes and edges with no cycles
- Root: A node with no parent
- Descendants: All of the nodes that can be reached by some path starting from a given node
- Ancestors: All of the nodes on the path from a node to the root
- Leaves: Nodes without children
- Siblings: Nodes of the same depth
- Parent: The immediate predecessor of a given node
- Child: The immediate successor(s) of a given node
- Depth: The length of the path from the given node to the root
- Node height: The length of the path from the given node to the deepest leaf
- Tree height: The height of the root
- Full tree: A tree in which every node other than the leaves has the maximum number of children
- Complete tree: A tree that is completely filled
- Nearly complete tree: A tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right and has no missing nodes
- Perfect tree: A full tree in which all leaves have the same depth
- Balanced tree: (With respect to a binary tree) a tree in which the height of the left and right sub-trees of every node differ by 1 or 0 (as used by an AVL tree)

27. Given a tree, identify the [root | leaves | tree height | tree size]. Given a tree, label each node with its [depth | height]. Given a tree, identify if it is [full | (nearly) complete | perfect | balanced].

    Review class lectures for examples.

28. Explain how to store a binary tree using an [array | linked structure].

    (a) For an array, explain how to find the child/parent of a given node based only on the index of that element.

    (b) For a linked structure, write the code for a node.

    Can you generalize the idea to k-ary trees

    Revisit later.

29. Write a recursive method to calculate the [size | height] of a tree.

    Waiting on hearing if sharing these would be an honor code violation.

30. Given a tree, perform a [pre-order | post-order | in-order | level-order] walk of the tree, printing the nodes.

    Review class lectures for examples.

31. Write code for a recursive [pre-order | post-order | in-order] tree traversal of a binary tree that prints the node values out. Assume a generic `Node<T>` class with a data field and a array of node children.

    Waiting on hearing if sharing these would be an honor code violation.

3.5 Hashing

32. What is the goal of hashing? (i.e. Why would we want to hash something into a hash table?)

33. Explain the difference between a hash code, a hash function, and a hash table. How does a hash code differ from the index of a hash table?

34. Create a hash function for [strings | numbers | cards | people] which generates relatively unique values.

    Waiting on hearing if sharing these would be an honor code violation.

35. Explain Java's hash-contract and why it is necessary.

36. Explain the idea of "load" on a table and how it relates to the big-O of the hash table's [`add()` — `remove()` | `contains()`] methods.

37. Hash [strings | numbers | cards | people] into a hash table of size [some number] using separate chaining and the hash function you wrote above.

    Waiting on hearing if sharing these would be an honor code violation.

38. Rehash your tables above to [a prime number over double the size].

39. Write code for an [`add()` | `remove()` | `contains()` | `rehash()` | `getLoad()`] method in a hash table which uses separate chaining.

    Waiting on hearing if sharing these would be an honor code violation.

3.6 General Programming and Java Specific

40. The concatenation operator takes $O(n^2)$ time since $\exists c : n = mc \implies O(nm) = O(c{\cdot}n^2)$ and $O(c{\cdot}n^2) \in O(n^2)$.

41. A generic class is defined by a *type parameter*. If we wanted to create a box that could hold any type of data (borrowing from the first lecture), we would have `class Box<T> { T data; }`.

    Generic classes are useful for data structures because they reduce code duplication and encourage code-reuse. Why make a box for a `char`, `short`, `int`, and `long` when you can make one box that holds all kinds of data? The unspoken benefit is that it also makes your data structure easy to extend – for instance, the `Box` object can hold things that might be beyond the scope of the original author's imagined use cases.

42. Explain recursion to someone who has never heard of it before. Make sure your answer includes the following terms: base case, recursive case, readability, efficiency, and iteration.

    Reminds me of work so I'll revisit this later.

43. Given an iterative function, write the same function using recursion. Given a recursive function, write the same function using iteration.

    Review class lectures for examples.

44. Explain how you can walk backward in a singly linked list using either recursion or a stack. Why does recursion work without needing to make your own stack?

    Revisit later.