Part 1: Data Structure Trade-offs (20 points total)

Thus far in class, we have discussed several data structures: The `Array` and `LinkedList`, variants of the `Buffer`, `Stack`, and `Queue` ADTs based on arrays and linked lists, and the `EdgeList` and `AdjacencyList` structures. Consider each of the following situations, and identify a specific data structure that you would use in the situation. In *at most* one sentence for each, identify one or two ADT operations that are relevant to the use case, and state their complexity in the selected data structure.

Question 1 (10 points)

The data structure encodes a collection of users to which new users are constantly being added. Users are identified by their position in order in which they were added, and we need to access users by their position quickly.

Answer:
We want adding new users to the end of the list (`append`) to be efficient, which means we want a `Buffer`. Since we want efficient positional access to elements (`apply`) of the `Buffer`, we need an `ArrayBuffer`.

- ☐ 4 pt: Answer identifies the need for (potentially amortized) constant-time append, references the `Buffer` ADT, or identifies a data structure (Linked List, ArrayBuffer, Dequeue) that has this property.
- ☐ 3 pt: Answer identifies the need for constant-time access by position, or references a data structure (Array, Dequeue, or any Array-based structure) that has this property.
- ☐ 3 pt: Answer identifies need for an `ArrayBuffer` specifically.

Question 2 (5 points)

The data structure encodes a road network consisting of intersections and roads between them. We want to be able to find the shortest path between any two intersections.

Answer:
The data we want to store is a graph, so we want a data structure designed specifically to store that. Of the `EdgeList` and `AdjacencyList` structures, the latter is strictly better.

- ☐ 3 pt: Answer identifies the need for a graph data structure.
- ☐ 1 pt: Answer references either the `EdgeList` or the `AdjacencyList` structure.
- ☐ 1 pt: Answer identifies `AdjacencyList` structure.

Question 3 (5 points)

A todo list where we delete items from the list as we cross them off. It is very important that adding a single item to the list be fast (no lag spikes). [[**Correction: Assume that items will always be removed from the todo list in the order in which they were inserted.**]]

Answer:
FIFO order requires a `Queue`. Of the two queue implementations, we care about the cost of an individual insertion ("no lag spikes"), so the Amortized $O(1)$ runtime of an `ArrayQueue` is not acceptable. We want a `LinkedListQueue`.

- ☐ 2 pt: Answer identifies the need for a `Queue`.
- ☐ 2 pt: Answer identifies the need for a linked list, or calls out the inadequacy of amortized $O(1)$ runtime for this need.
- ☐ 1 pt: Answer identifies the need for a linked-list based `Queue` specifically.

Part 2: Bounds (20 points total)

For each of the following equations, state the (closed form) tight upper and lower bounds (i.e., Big-O and Big-Ω bounds) in terms of a complexity class. If a tight bound (i.e., Big-Θ) exists, state this as well.

Question 4 (4 points)

$$f_1(n) = \sum_{i=0}^{\log(n)} 2^i \cdot \frac{n}{2^i}$$

Answer:

The 2^is cancel out, and n is independent of i, so the result is $\sum_{i=0}^{\log(n)} n = \Theta(n \log(n))$

1. **Big-O:** $n \log(n)$
2. **Big-Ω:** $n \log(n)$
3. **Big-Θ:** $n \log(n)$

☐ 2 pt: Answer has identical values for all three parts of this problem.

☐ 2 pt: Answer for Big-Θ is $n \log(n)$.

Question 5 (4 points)

Variant 1:

$$f_2(n) = 5n^2 + \log(2^n) + 6n$$

Variant 2:

$$f_2(n) = 2^{\log(n)} + 6n^3 + n\log(n)$$

Variant 3:

$$f_2(n) = 5n^3 + \log(2^n) + 6n$$

Variant 4:

$$f_2(n) = 2^{\log(n)} + 6n^4 + n\log(n)$$

Answer:

1. **Big-O:** n^2 or n^3 or n^3 or n^4 (respectively)
2. **Big-Ω:** same as above.
3. **Big-Θ:** same as above.

☐ 2 pt: Answer has identical values for all three parts of this problem.

☐ 2 pt: Answer for Big-Θ is (by variant) n^2 or n^3 or n^3 or n^4.

Question 6 (4 points)

Variant 1:

$$f_3(n) = \sum_{i=1}^{n} (2i + 3 + 7n)$$

Variant 2:

$$f_3(n) = \sum_{i=1}^{n^2} (3i + 2 + 7n)$$

Variant 3:

$$f_3(n) = \sum_{i=1}^{\sqrt{n}} (7i + 3n + 2)$$

Variant 4:

$$f_3(n) = \sum_{i=1}^{2^n} (7i + 2n + 3)$$

Answer:

This question tests summations. Addition commutes through summation, so all variants reduce to some form of $\left(\sum_{i=j}^{k} c_1 i\right) + \left(\sum_{i=j}^{k} c_2\right) + \left(\sum_{i=j}^{k} c_3 n\right)$ that can each be handled individually. The sum of i is $\Theta(k^2)$ (for any constant j), the sum of a constant is $\Theta(k)$, and the sum of a constant times n is $O(kn)$. From there, it remains to pick the dominant term.

1. **Big-O:** n^2 or n^4 or $n\sqrt{n}$ or $(2^n)^2$ (respectively)
2. **Big-Ω:** same as above.
3. **Big-Θ:** same as above.

☐ 2 pt: Answer has identical values for all three parts of this problem.

☐ 2 pt: Answer for Big-Θ is (by variant) n^2 or n^4 or $n\sqrt{n}$ or $(2^n)^2$ (1 pt for an answer of n in variant 3).

Question 7 (4 points)

$$f_4(n) = \sum_{i=1}^{4} \sqrt{i}$$

Answer:
This question is taken directly from the homework. As it was in the homework, the key to the solution is the fact that the upper and lower bounds of the summation are constant.

1. **Big-O:** 1
2. **Big-Ω:** same as above.
3. **Big-Θ:** same as above.

☐ 2 pt: Answer has identical values for all three parts of this problem.

☐ 2 pt: Answer states any constant value for Big-Θ.

Question 8 (4 points)

$$f_5(n) = \begin{cases} \textbf{[v1:}(\log(n))\textbf{ or v2:}(1)\textbf{ or v3:}(n^2)\textbf{ or v4:}(n\log(n))\textbf{]} & \textbf{if } n \text{ is odd} \\ \textbf{[v1:}(n^2)\textbf{ or v2:}(n)\textbf{ or v3:}(n^3)\textbf{ or v4:}(n^2)\textbf{]} & \textbf{if } n \text{ is even} \end{cases}$$

Answer:
Regardless of variant, this function spans multiple complexity bounds. The higher class is the Big-O, the lower class is the Big-Ω, and as a consequence, there is no Big-Θ.

1. **Big-O:** n^2, or n, or n^3, or n^2, respectively.
2. **Big-Ω:** $\log(n)$, or 1, or n^2, or $n\log(n)$
3. **Big-Θ:** Not defined.

☐ 2 pt: Answer notes that there is no Big-Θ bound.

☐ 1 pt: Answer for Big-O is (by variant) n^2, or n, or n^3, or n^2.

☐ 1 pt: Answer for Big-Ω is (by variant) $\log(n)$, or 1, or n^2, or $n\log(n)$.

Part 3: Data Structure Design (20 points total)

You are implementing the data structure supporting an operating system's task scheduler. This structure tracks the tasks that need to be scheduled, defaulting to FIFO order. The data structure should behave like a queue with efficient[1] **enqueue** and **dequeue** operations. Additionally, the structure needs to support two additional operations *efficiently*:

- **bump**: Cause a task to advance one spot in the queue, skipping ahead of the task immediately ahead of it.
- **remove**: Immediately remove a task from the queue, regardless of its position.

The operating system already keeps state for each task in a *task struct*. You may add additional fields to the task struct, and assume that the task struct (and therefore any state that you wish to associate with the task) is passed in mutable form to the **enqueue**, **bump**, and **remove** methods. **[[Correction: Think of the task struct as a task "class." You can add whatever fields you like to it, just list them in your answer.]]**

Question 9 (20 points)

Outline the design of a data structure that supports efficient (i.e., $O(1)$ or amortized $O(1)$) **enqueue**, **dequeue**, **bump**, and **remove** operations. Provide a short 1-3 sentence answer that specifically addresses the following points: (i) The underlying data structure on which your queue will be based, (ii) What additional state you would add to the task struct, (iii) when and how will this state be manipulated, and (iv) how this state is used by the **bump** and **remove** methods.

Answer:

The criteria being requested are met by an unsorted variant of the linked list structure that was implemented in PA2; This is a good starting point, but not required to answer the question.

Since we're talking about FIFO order, we probably want a `Queue`, which leaves open the question of whether we want an `Array` or Linked `List` based queue. The former can't efficiently handle **insert** or **remove** operations (except at the end), while the latter only has constant-time operations at the head and tail, or by-reference.

Per the requirements, we need to support **remove** efficiently, so `Array` is not an option. With that, the question becomes how we support access by reference efficiently. The task struct helps us here, since we can put a reference to the linked list node there (i.e., like PA2's main function, or the `EdgeList`/`ArrayList` structures).

A reasonable answer to this question has the form: I would use a linked-list based queue. By storing a reference to the linked list node corresponding to a task in the task struct, we can implement **bump** and **remove** in constant time through access-by-reference.

- ☐ 5 pt: Answer includes a reference to `Queue`
- ☐ 5 pt: Answer chooses a linked list or otherwise communicates that an `Array`-based structure is not viable.
- ☐ 5 pt: Answer indicates that a reference to the linked list node should be stored in the task struct.
- ☐ 5 pt: Answer indicates that the reason for having a reference to the linked list is to implement efficient **remove** and/or **bump**, or that it allows $O(1)$ runtime operations on the linked list.

[1] Unless otherwise noted assume an efficient operation is defined as an operation with a runtime complexity of $O(1)$ or amortized $O(1)$.

Part 4: Short Answer (20 points total)

Question 10 (10 points)

You are working at a social media company, and receiving *infrequent* reports from users that it takes forever to
Variant 1: ... make a post. You track the issue down to the insert method of an ArrayBuffer being used for a queue in the system.
Variant 2: ... [[**Correction: load their feed**]] . Later that day, you hear rumors of a similar issue infrequently affecting users load their feed. You track the issue down to a part of the code that sorts the past week of posts in the user's feed. Even accounting for the number of posts, a small number of these sorts take unusually long.
Variant 3: ... make a post. You track the issue down to the insert method of an ArrayBuffer being used for a queue in the system.
Variant 4: ... [[**Correction: load their feed**]] . Later that day, you hear rumors of a similar issue infrequently affecting users load their feed. You track the issue down to a part of the code that sorts the past week of posts in the user's feed. Even accounting for the number of posts, a small number of these sorts take unusually long.

- In at most one sentence, what is the most likely cause (given the knowledge you have learned CSE-250)?
- In at most one sentence, propose a fix.

Answer:
Both of these questions are asking about operations with non-guaranteed runtimes (*amortized* $O(1)$ for a single call to `ArrayBuffer`'s **append**; or *expected* $O(n\log(n))$ for a call to `QuickSort`). Amortized runtimes give you guarantees about multiple calls, but not individual calls. Similarly, expected runtimes give you no guarantees at all. Since the events are infrequent, the most likely diagnosis is that we're hitting corner cases for these algorithms.
The solutions are, respectively, to replace the structure/algorithm with a comparable algorithm with an unqualified (i.e., guaranteed) runtime bound. For example, `ArrayBuffer` could be replaced with a Linked `ListBuffer`, or `QuickSort` could be replaced with a `MergeSort`.

- ☐ 7 pt: Answer communicates an understanding of the fact that amortized/expected runtimes do not provide guarantees for individual calls.
- ☐ 2 pt: Answer communicates the need to use an algorithm with unqualified runtime bounds.
- ☐ 1 pt: Answer provides a specific, correct example of an algorithm with unqualified runtime bounds.

Question 11 (10 points)

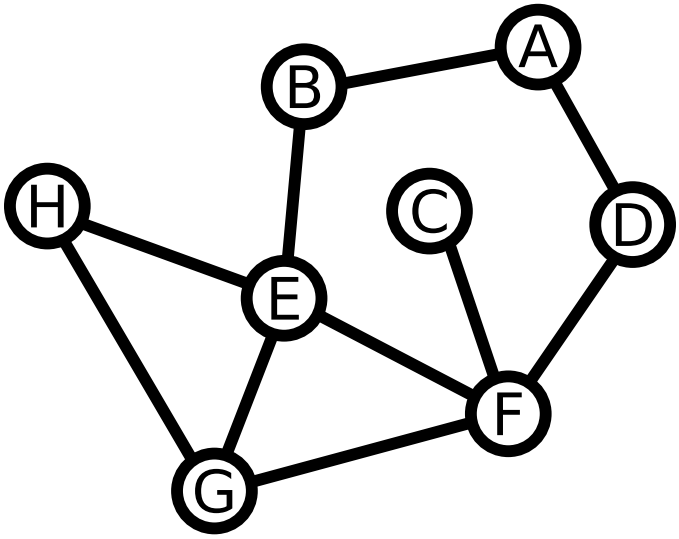
Consider an array of size 100, where each record is **Variant 1:** 8
Variant 2: 4
Variant 3: 2
Variant 4: 16 bytes long. Assume that the 0th element of the array is located at memory address A. At what position is the **Variant 1:** 10th
Variant 2: 20th
Variant 3: 25th
Variant 4: 10th element located. **[[Correction: To clarify, the question is asking at what memory address the element is located at?]]**

Answer:
In an Array with fields of size c, with the first element located at address A, element i will be located at $A + c \cdot i$. The answers are respectively $A + 80$, $A + 80$, $A + 50$, and $A + 160$.

- ☐ 5 pt: Answer communicates an understanding of array positions located at $A + c \cdot i$
- ☐ 5 pt: Correct answer, or sufficient work to indicate that the answer would have been correct if not for a math error.

Part 5: Graphs (20 points total)

Consider the following graph G:



Question 12 (20 points)

Draw or otherwise identify a (spanning) subgraph S that has the following properties:

- S is a spanning subtree
- [**v1:(** S contains no vertices with degree greater than 2. **)** **or v2:(** S contains no vertices with degree greater than 2. **)** **or v3:(** S contains at least 2 vertices with degree greater than 3. **)** **or v4:(** S contains at least 2 vertices with degree greater than 3. **)**]

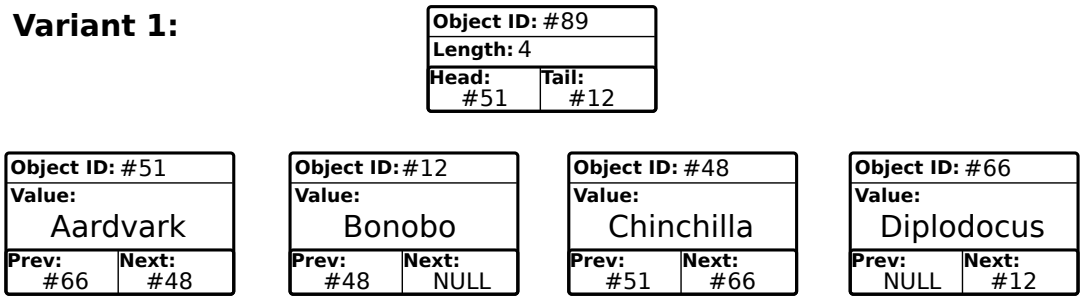
Answer:
A spanning subtree has exactly one path between any pair of nodes. There are a few different examples of spanning subtrees that meet the above criteria, but the key points are: (i) Not having a degree greater than 2 means a line, so for example CFDABEGH; and (ii) The three nodes that could potentially have degrees greater than 3 are E, G, and F. The criterion is satisfied if E is connected to H, G, B, and F, and if F is connected to E, C, and D

- ☐ 7 pt: Answer is a spanning subgraph (visits every node).
- ☐ 7 pt: Answer is a tree (no cycles).
- ☐ 6 pt: Answer meets the variant-specific criteria (line or 2 nodes with 3 degree)

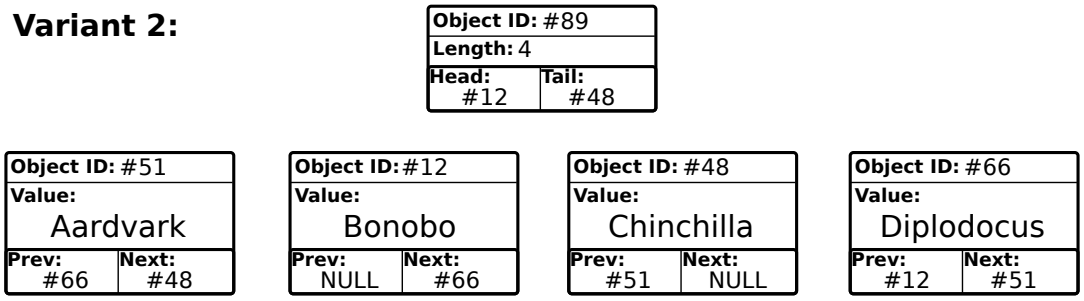
Part 6: Bonus: Debugging (5 points total)

Consider the following diagram, showing the object representing a doubly linked list and several of its nodes.

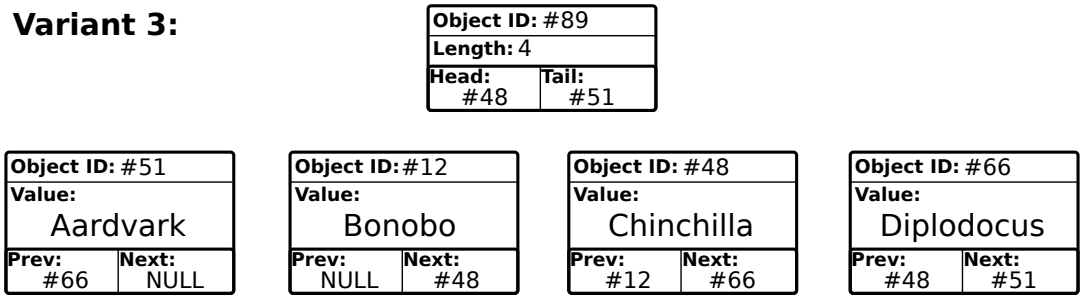
Variant 1:



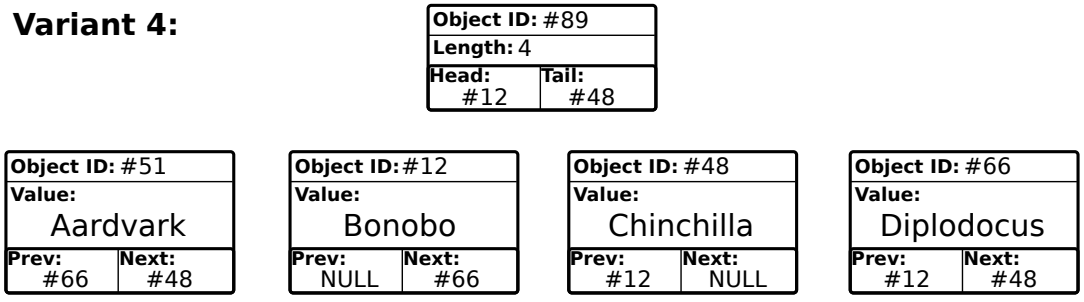
Variant 2:



Variant 3:



Variant 4:



Question 13 (5 points)

Identify the error in the linked list structure above by identifying the field(s) that is(are) inconsistent.

Answer:
On variant 1, the head/next and tail/prev pointers define unrelated orders (A, C, D, B vs D, A, C, B).
On variant 2, the linked list is correct.
On variant 3, the head node points to the 2nd node, not the first.
On variant 4, the list only consists of 3 items (i.e., the length field is wrong).

- ☐ 5 pt: The bug is identified ((i) unrelated forward/backward orders, (ii) correct list, (iii) head points to 2nd node, (iv) list length is wrong).