Part A: Linked Set (10 points total)

When discussing Hash Maps, we briefly discussed the Linked Hash Map structure, which combines a Linked List with a Hash Map to provide linear-time (in `n`, rather than `N`) iteration of the map's contents. The Linked Map structure has a few other useful tricks as well.

Consider the analogous Linked Set structure, summarized below:

```
class LinkedSet[A] extends mutable.Set[A]
{
  val insertOrder = DoublyLinkedList[A]
  val contents = mutable.Map[A, DoublyLinkedListNode[A]]()

  def add(a: A): Unit =
  {
    if(contents.contains(a)){ return }
    val node = insertOrder.prepend(a)
    contents.put(a, node)
  }
  def apply(x: A): Boolean =
  {
    contents.contains(x)
  }
  def remove(x: A): Unit =
  {
    contents.remove(x) match {
      case Some(node) => node.removeSelfFromList() /* Remove by position */
      case None => /* no-op */
    }
  }
  def removeOldest = ???
}
```

Question A.1 (4 points)

Provide a pseudocode implementation of the `removeOldest()` function that removes the oldest item still in the set. This is the item that was least recently added (ignoring duplicate additions). Your implementation should have the same complexity as `contents.remove()`.

Answer

```
def removeOldest(): Unit =
  remove(insertOrder.last)
```

Question A.2 (6 points)

Assume the variable `contents` is implemented as a `TreeMap` or a `HashMap` (with chaining and a constant α_{max}), respectively. Provide tight worst-case, amortized, and expected runtime bounds for the `add` method for each variant in terms of n (i.e., in terms of `contents.size`).

	`TreeMap`	`HashMap`
Worst-Case	**Answer** $O(\log(n))$	**Answer** $O(n)$
Amortized	**Answer** $O(\log(n))$	**Answer** $O(n)$
Expected	**Answer** $O(\log(n))$	**Answer** $O(1)$

Part B: Project Review (20 points total)

Question B.1 (10 points)

Consider a Log-Structured Merge Index data structure (i.e., as in PA2) with a buffer size of 2 (i.e., a promote happens every 2 insertions, $O(2^\ell)$ records at level ℓ). The following keys are inserted into the data structure:

15, 24, 10, 4, 25, 2, 12

Draw the state of the data structure (i) after every promote step, and (ii) its final state.

Answer

1. **Level 1**: [15, 24]
2. **Level 1** Empty; **Level 2**: [4, 10, 15, 24]
3. **Level 1**: [2, 25]; **Level 2**: [4, 10, 15, 24]

Final State:

Buffer: [12]; **Level 1**: [2, 25]; **Level 2**: [4, 10, 15, 24]

Question B.2 (10 points)

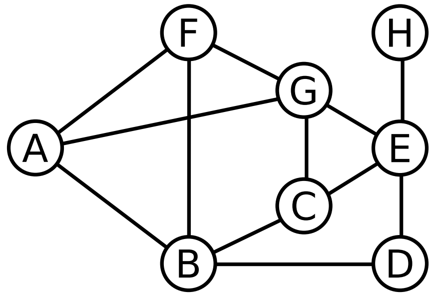
Campaign contributions are public data. Suppose each record in this data set corresponds to a single donation, and includes at least the following three attributes: (i) the name of a politician, (ii) the amount of money that was donated, and (iii) the name of the donating entity. Write pseudocode for an algorithm to compute the *total* amount donated *to each politician* . Assume that each politician has a unique name, and that the name is used consistently throughout the data set. Your algorithm must have a runtime that is linear ($O(n)$) in the number of records. Be sure to note which specific data structure(s) your algorithm uses.

Answer

```
def total(donations: Seq[Donation]): Map[String, Double] =
{
  val totals = mutable.HashMap[String, Double]()
  for(d ← donations)
  {
    totals(d.politician) =
      d.amount + totals.getOrElse(d.politician, 0.0)
  }
  return totals
}
```

Part C: Graphs (15 points total)

The following questions pertain to the following graph



Question C.1 (5 points)

List the nodes visited by Depth First Search in the order in which they are visited, starting from vertex `B`. Assume that edges are explored in alphabetical order of the opposite vertex. For example, from vertex `D`, edge `DB` would be explored before edge `DE`.

Answer

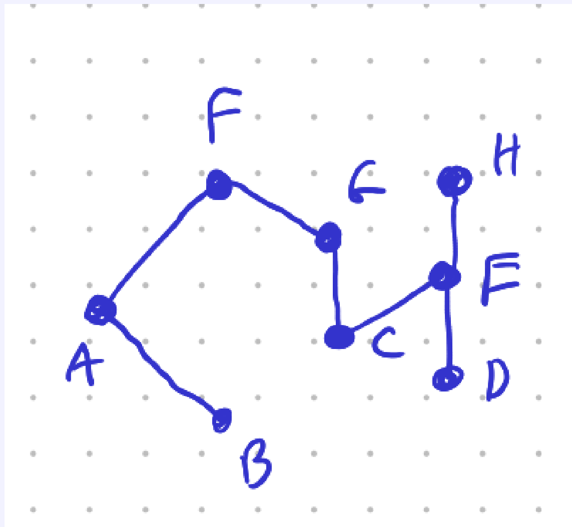
1. **B**
2. **A** (Explanation: Started at B and adjacent edges are A, C, D)
3. **F** (Explanation: Last step was to A and adjacent edges are B, F, G; B is already visited)
4. **G** (Explanation: Last step was to F and adjacent edges are A, B, G; A, B are already visited)
5. **C** (Explanation: Last step was to C and adjacent edges are F, A, C, E; F, A are already visited)
6. **E** (Explanation: Last step was to E and adjacent edges are B, G, E; B, G are already visited)
7. **D** (Explanation: Last step was to D and adjacent edges are C, D, G, H; C is already visited)
8. **H** (Explanation: Last step backtracked to E and and adjacent edges are C, D, G, H; C, D, G are already visited)

(Note: Explanations are provided as part of the answer key, but not a required part of the answer)

Question C.2 (5 points)

Draw the spanning tree produced by DFS on the graph above.

Answer



Question C.3 (5 points)

Assume that the graph above is stored using an adjacency list data structure. Draw the vertex list from this data structure. Your illustration does not need to precisely capture the pointer structure of the list contained at each vertex, but should at least state which edges are associated with each vertex.

Answer

- $A \rightarrow [AB, AF, AG]$
- $B \rightarrow [AB, BF, BC, BD]$
- $C \rightarrow [BC, CG, CE]$
- $D \rightarrow [BD, DE]$
- $E \rightarrow [CE, DE, EG, EH]$
- $F \rightarrow [AF, BF, FG]$
- $G \rightarrow [AG, CG, EG, FG]$
- $H \rightarrow [EH]$

Part D: Sets And Maps (10 points total)

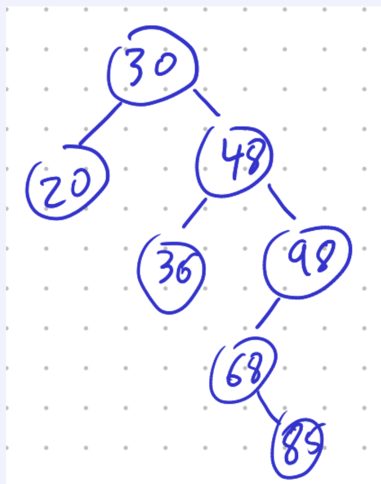
The following items are inserted into a set (in the order given):

30, 20, 48, 36, 98, 68, 85

Question D.1 (3 points)

Show the final structure resulting after the items above are inserted if the set is stored as a general binary search tree.

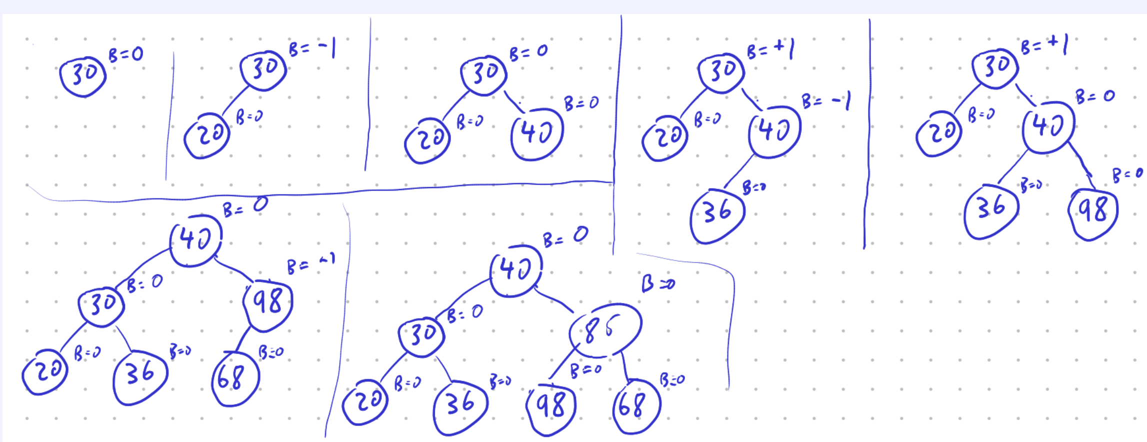
Answer



Question D.2 (3 points)

Show the final structure resulting after the items above are inserted if the set is stored as an AVL tree. *Show the tree after each insertion.*

Answer



Question D.3 (4 points)

Show the final structure resulting after the items above are inserted if the set is stored as a hash table with chaining. *Show the structure just prior to every rehash operation.* Use an initial capacity of 5 and $\alpha_{max} = 0.5$, along with the following hash function:

hash(20)=69 ; hash(30)=22 ; hash(36)=39 ; hash(48)=27 ; hash(68)=72 ; hash(85)=49 ; hash(98)=61

Answer

Rehash At

0	1	2	3	4
		30		20

$n > 2.5$

0	1	2	3	4	5	6	7	8	9
	98	30					48		20 → 36

$n > 5$

0	1	2	3	4	5	6	7	8	9
	98	30					48		20 → 85

10	11	12	13	14	15	16	17	18	19
		68							36

$n > 10$

Part E: Short Answer (30 points total)

Question E.1 (5 points)

Explain how to create an **immutable** queue with amortized constant ($O(1)$) time enqueue and dequeue operations. Recall that changes to immutable structures require constructing a new object that encodes the modified version of the data structure. The new object may re-use components of the original version.

Answer

Use two singly linked lists. Call them 'ready' and 'pending'.

The `prepend`, `tail`, and `head` operations (i.e., operations that act only on the head) on an *immutable* singly linked list are $O(1)$. (Since the list nodes are immutable, they can safely be re-used across multiple versions of the linked list)

`enqueue` is implemented by prepending to the pending list, which is $O(1)$, as noted above.

`dequeue` is implemented in two steps:

- If the ready list is empty, pop every element (One call to `head` and `tail` each, so $O(1)$ per element) off of the pending list and insert it into the ready list (in reverse order). This is $O(|pending|)$.
- Pop the head off the ready list and return it ($O(1)$).

Note that, although items are added to the pending list in reverse order, the order is reversed as they are added to the ready list. When `dequeue` returns the head of the ready list, it is the least recently added item.

Observe that a constant amount of work is performed per element in `dequeue`:

- The item is added to the head of the pending list $O(1)$
- The item is removed from the head of the pending list $O(1)$ and added to the head of the ready list $O(1)$.
- The item is removed from the head of the ready list $O(1)$

Thus, the total work required for n calls to any interleaving of `enqueue` and `dequeue` is always $O(n)$, and so the amortized per-call cost of these operations is $O(1)$.

(**Note**: This question was provided to the class a week prior to the final exam)

Question E.2 (5 points)

Show an example of a function $f(n)$ for which both $f(n) \in \Omega(n\log(n))$ and $f(n) \cdot n \in \Omega(n\log(n))$

Answer

Answer: $n\log(n)$ or any function in the same or higher complexity class.

Explanation: $\Omega(g(n))$ is the set of all functions in the same or greater complexity class as g. $f(n) \cdot n \in \Omega(n\log(n))$ is the same as saying that $f(n) \in \Omega(\log(n))$, so what the question is asking for, is a function that is both in $\Omega(n\log(n))$ and $\Omega(\log(n))$. Since $n\log(n) \in \Omega(\log(n))$, but not visa versa, the question can further be simplified to any function in the same or greater complexity class as $n\log(n)$.

Question E.3 (5 points)

Consider an implementation of QuickSort that consistently uses the first item in the range being sorted as a pivot. For example, when partitioning the range $[4, 8)$, the value at index 4 would be used as a pivot. Explain how to construct an input that, when used with this implementation of QuickSort, will consistently trigger QuickSort's worst case runtime of $O(n^2)$. Be sure to explain why the resulting input would result in the worst-case runtime.

Answer

Answer: Any sequence already sorted in ascending or descending order. (This is the simplest answer, more intricate, but still correct answers also exist)

Explanation: QuickSort's expected performance assumes that the pivot value will be chosen, on average, near the median value of the sequence. To be technical: It assumes that the *expectation* of the position of the randomly selected value is $\frac{n}{2}$. This assumption holds if we pick an element at a random position as a pivot. However, in this example, the pivot selection method is deterministic, not random. The worst case $\Theta(n^2)$ behavior takes place if we consistently pick a value at (or close to) the 0th or nth position (because the recursive step will sort two lists of size 0 and $n-1$ respectively). Again, technically, to get this worst case behavior, we want the expectation of the position of the pivot to be (close to) 0 or n.

Since the first element is always chosen as a pivot, we want it to be the 0th (lowest) or nth (greatest) rank element.

Question E.4 (5 points)

Say you have an unsorted sequence. You could (i) use a linear scan to find a value, or you could (ii) sort it and then use binary search to find the value. For each of these two approaches, for what sorts of use cases is the approach better than the other. Your response should contrast of the runtime complexity of each approach.

Answer

The runtime complexity of a linear scan is $O(n)$; Sorting is $O(n \log(n))$ (with the best-known algorithms), and binary search is $O(\log(n))$. Thus, method (i) is $O(n)$, while method (ii) is $O(n \log(n) + \log(n)) = O(n \log(n))$.

The first method is better if you're only looking for one value.

The second method can be better if you're going to be looking for multiple values. The $O(n \log(n))$ cost only needs to be paid once, and every subsequent lookup will be $O(\log(n))$.

Note: Another acceptable answer is that the second method can also be better if you have advance notice and can sort the list before you need the value.

Question E.5 (5 points)

Assume you have a B+Tree with n records. Assume that C is the capacity of both data and directory pages (i.e., the number of key/pointer pairs and the number of records that fit on one data page). Recall that the `range(low, high)` method on an ordered map returns the records with keys in the range $[low, high)$. Assume that there are k such keys. What is the worst-case (i.e., big-O) IO complexity, in terms of n and k, of this method.

Answer

There are two factors in this operation: finding one end of the range (e.g., low) and then iterating over all of the k records.

Finding one end of the range is a normal $B+$ tree traversal: $O(\log_C(n))$ disk reads.

Iterating over k records requires iterating over the subtree containing the k records. There are $O(\frac{k}{C})$ data pages we need to iterate over, and the subtree containing these pages will have $O(\frac{k}{C})$ nodes.

Answer: $O(\log_C(n) + \frac{k}{C})$

Question E.6 (5 points)

Nearly every set of slides for Dr. Kennedy's CSE-250 lectures features a picture on the title page that is not directly related to CSE-250 itself. What are the pictures of?

Answer

Cats (Julie and Serenity)

Part F: Asymptotic Bounds (15 points total)

Question F.1 (15 points)

For each of the following functions, provide a tight Big-O, Big-Ω, and Big-Θ bound, or indicate that the bound does not exist.

$$f(n) = 2^{\log_2(4n)} + 19n\log(n) + 12n^2$$

$$g(n) = \sum_{i=0}^{n^2} i$$

$$h(n) = \begin{cases} 2^n & \textbf{if } n \text{ is odd} \\ n^2 & \textbf{if } n \text{ is even} \end{cases}$$

	Big-O	**Big-Ω**	**Big-θ**
$f(n)$	**Answer** n^2	**Answer** n^2	**Answer** n^2
$g(n)$	**Answer** n^4	**Answer** n^4	**Answer** n^4
$h(n)$	**Answer** 2^n	**Answer** n^2	**Answer** n/a