Writing Solutions for Design Problems

- good solutions typically contain
  - the high-level idea
    - explain the approach before describing operations
  - the representation
    - describe what is stored and how it is organized
  - the operations
    - explain how each operation works
  - a correctness justification
    - explain why the structure maintains the required properties
  - a runtime analysis
    - explain why each operation meets the required complexity

Writing Solutions for Design Problems

- three key points
  - start with the idea, not the implementation details
  - choose the right levels of detail and abstraction
  - include (only) the details that matter

The Right Levels of Detail and Abstraction

- there are multiple ways to describe processes (operations and algorithms)
  - prose
  - structured English
  - pseudocode
- the most appropriate choice depends on
  - the complexity of the operation
  - the level of abstraction that is possible
  - whether correctness or running time depends on specific steps

The Right Levels of Detail and Abstraction

- use prose when
  - the operations are simple
  - standard structures are used in straightforward ways
  - name things for clarity when pseudocode isn't otherwise needed
- use pseudocode when
  - correctness depends on specific steps or case handling
  - multiple things interact – precision is needed for clarity
  - it is simpler and shorter to write the necessary details in pseudocode
    - don't describe code in words

The Right Levels of Detail and Abstraction

- do not explain –
  - basic data structures (arrays, stacks, binary search trees, hashtables, …)
  - standard operations and well-known algorithms

What Details Matter?

- include details when they affect or are needed for assessing correctness and running time
- for correctness
  - can someone with a standard amount of knowledge carry it out?
  - is a well-defined result sufficient, or is the process what matters?
- for running time
  - is how to carry it out in the desired running time standard knowledge?
  - a well-defined result is *not* sufficient when the running time depends on the specific process

What Details Matter?

- process vs well-defined result
  - e.g. delete *x* from a binary search tree
    - find the node containing *x*
    - if *x* does not have any leaf children, find the node containing *x*'s successor and swap the elements
    - remove the node (now) containing *x*
  - "find the successor" is well-defined and sufficient for correctness, but describing the process of *how* the successor is found is necessary for running time

Discussion #1

- design a data structure that supports
  - `insert(x)` – O(1)
  - `findMin()` – O(1)
- proposed solution

  Store the elements in a linked list.
  Maintain a pointer to the minimum element.
  When inserting a new element, compare it to the current minimum and update the pointer if necessary.

Discussion #1

- assessment – not bad, but not ideal
  - overview
    - not explicitly called out, but the idea is contained within the first few sentences without much additional detail to get in the way
  - representation
    - not explicitly called out, but the first two sentences identify the necessary structures without much additional detail to get in the way
  - operations
    - the reader is left to figure out some of the operations (insert includes inserting into the linked list, findMin makes use of the min pointer) though those are pretty standard/straightforward
  - correctness
    - not explicitly stated, but it is straightforward for the reader to assess for themselves that the min pointer is updated correctly
  - running time
    - not addressed, though all of the operations are pretty standard and straightforward for the reader to assess
  - level of detail
    - that the elements are contained in a linked list specifically is not important for either correctness or running time – O(1) insert is well-known for both arrays and linked lists, and both have O(1) locators (array index, pointer to list node)

Discussion #1

- a better solution
  - leave out the irrelevant implementation details (linked list)
  - address running time
  - explicitly include the other elements

Idea: maintain a reference to the minimum element

Store: list of elements, reference to the minimum element

Operations:
– insert(x) – add the element to the list, update the min element to *x* if *x* is smaller than the current min
– findMin() – return the stored minimum

Running time: both operations are O(1) as required
– insert – adding to a list is O(1) (add to the end of an array or the head of a linked list) and updating the min element involves only a comparison and an assignment
– findMin – just returns a stored value

Discussion #2

- design a data structure that supports
  - `insert(x)` – O(log n)
  - `deleteMin(x)` – O(log n)
- proposed solutions

Use a priority queue to store the elements. Insertions and deletions use the priority queue operations.

Store the elements in a heap.
– insert(x): add the element to the heap and restore the heap property using the standard heap insert
– deleteMin(): remove the root element and restore the heap property using the standard heap remove min
A heap supports both operations in O(log n) time.

```
insert(x):
  place x at the next available leaf position
  i = index of new node
  while i > 1 and A[parent(i)] > A[i]
    swap A[i] with A[parent(i)]
    i = parent(i)
```

(with similar pseudocode for deleteMin)

Discussion #2

- solution #1 assessment – pretty good, but running time should be addressed
  - overview
    - stated in the first sentence
  - representation
    - covered in the first sentence
  - operations
    - covered in the second sentence
  - correctness
    - not explicitly stated, but insert and deleteMin are exactly the PQ operations
  - running time
    - not addressed but should be – O(log n) assumes a heap-based implementation and while that is the standard implementation of PQ, there are others so it is worth clarifying
  - level of detail
    - a standard PQ suffices and PQs are standard data structures so no need to go into greater detail

Discussion #2

- solution #2 assessment – also pretty good
  - overview
    - stated
  - representation
    - not explicitly called out, but covered by the high-level idea (just need a heap)
  - operations
    - addressed
  - correctness
    - not explicitly stated, but insert and deleteMin are exactly the PQ operations
  - running time
    - addressed – heap is identified, and insert and deleteMin are standard heap operations with known running times
  - level of detail
    - the description of the operations is overly detailed – insert and deleteMin are standard heap operations so no need to explain further
    - a higher level of abstraction is possible (PQ rather than heap), though since a heap implementation is important for the running time, directly specifying a heap is reasonable

Discussion #2

- solution #3 assessment – poor (too detailed and incomplete)
  - overview
    - none, just code
  - representation
    - not identified – have to sort through the code
  - operations
    - addressed
  - correctness
    - not addressed, and there's no overview to know what the operations are supposed to be doing, hindering understanding
  - running time
    - not addressed – `i` is not a simple counter and there is no overview to aid in understanding the loop in order to know how many times it repeats
  - level of detail
    - far too detailed – goes into implementation details of array-based heaps, when what is needed for correctness and running time is at most the conceptual idea of the heap ordering property and the bubbling-up operation to restore that property on delete

Discussion #2

- a better solution
  - a few tweaks to solutions #1-2

Idea: Store the elements in a heap.

Operations: insert(x) and deleteMin() can both be handled by the standard heap operations.

Running time: A heap supports both operations in O(log n) time.

Discussion #3

- design a data structure that supports
  - `enqueue(x)` – O(1)
  - `dequeue()` – O(1)
  - `findMax()` – O(1)

- proposed solution

Use a queue to store the elements.
Maintain a second deque that stores candidate maximum values in decreasing order.
When inserting a new element, remove smaller values from the back of the deque and append the new element.
When removing an element from the queue, also remove it from the front of the deque if it matches the maximum.

Discussion #3

- assessment – running time needs to be addressed
  - overview
    - not explicitly called out, but the idea is contained within the first few sentences without much additional detail to get in the way
  - representation
    - not explicitly called out, but the first two sentences identify the necessary structures without much additional detail to get in the way
  - operations
    - the reader is left to figure out some of the operations (findMax, that enqueue and dequeue also involve the queue) though it isn't a big leap
  - correctness
    - not explicitly stated, and probably takes working through an example for the reader to see why the idea works
  - running time
    - not addressed and potentially problematic – O(1) enqueue/dequeue is known, deques are standard though less common than queues, but insert calls for "removing smaller values" – how many? when a new maximum value is inserted, all of the values currently in the deque are smaller – it is not clear how this can be O(1)
  - level of detail
    - good – standard structures (queue, deque) are used at the level of the ADT without unnecessary implementation details