Chapter 12 Practice Exercises

Disclaimer: These practice questions are not official and may not have been vetted for course material. You may use these for practice, but you should prioritize official resources from current staff for a more accurate depiction of what you need to know for assignments and exams.

1. Which of the following statements is/are **TRUE** regarding amortization?
   I. Amortized complexity analysis typically results in a time complexity class that is worse than the asymptotic worst-case complexity of a given operation.
   II. Amortized complexity analysis is most useful when the average-case and worst-case time complexities of an operation are identical.
   III. Amortized complexity analysis provides a tighter upper bound on an operation for which individual calls may have different time complexities, regardless of input.

   A) I only
   B) II only
   C) III only
   D) I and III only
   E) II and III only

2. Suppose you are implementing a vector with an underlying dynamic array, but with a special insertion method known as `.shove_back()`. This method works normally when the array is not full (emulating `.push_back()`), but when the array fills up, the `.shove_back()` method creates a new dynamic array that is double the size of the old array, copies the elements from the old array to the new array, performs a $\Theta(n\log(n))$ call to `std::sort()` on the new array, and then deletes the old array. What is the amortized time complexity of the `.shove_back()` operation?
   A) $\Theta(1)$
   B) $\Theta(\log(n))$
   C) $\Theta(n)$
   D) $\Theta(n\log(n))$
   E) $\Theta(n^2)$

3. Suppose that we decide to modify the `.push_back()` method of the STL `std::vector<>` such that, instead of doubling the capacity during reallocation, we only increase the capacity by a factor of $\lceil 1.5\rceil$ (i.e., if a vector of size k reaches capacity, we move all the elements to a new vector of capacity $\lceil 1.5\rceil k$). Which of the following statements is/are **TRUE**?
   A) The amortized time complexity will be *higher (worse)* than when we doubled capacity during reallocation
   B) The amortized time complexity will be *lower (better)* than when we doubled capacity during reallocation
   C) The amortized time complexity will be *the same* as when we doubled capacity during reallocation
   D) More than one of the above
   E) None of the above

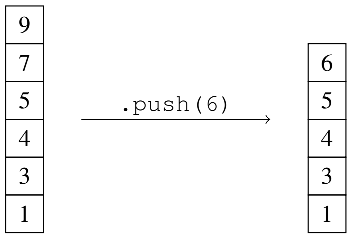
4. Suppose that we decide to modify the `.push_back()` method of the STL `std::vector<>` such that, instead of doubling the capacity during reallocation, we only increase the capacity by a constant value of 1000 with every reallocation (i.e., if a vector of size k reaches capacity, we move all the elements to a new vector of capacity $k+1000$). Which of the following statements is/are **TRUE**?
   A) The amortized time complexity will be *higher (worse)* than when we doubled capacity during reallocation
   B) The amortized time complexity will be *lower (better)* than when we doubled capacity during reallocation
   C) The amortized time complexity will be *the same* as when we doubled capacity during reallocation
   D) More than one of the above
   E) None of the above

5. Suppose you are performing a sequence of n operations on a data structures in which the i^{th} operation takes $\Theta(i)$ time if i is an exact power of 2 ($i = 1,2,4,8,16,32,\ldots$), and $\Theta(1)$ time otherwise. What is the amortized time complexity of a single operation on this data structure?
   A) $\Theta(1)$
   B) $\Theta(\log(n))$
   C) $\Theta(n)$
   D) $\Theta(n\log(n))$
   E) $\Theta\left(\frac{2^n}{n}\right)$

6. Consider an STL queue that is defined with a doubly-linked list as its underlying container (i.e., a `std::queue<T, std::list<T>>`). Which of the following correctly summarizes the worst-case and amortized time complexities of a single call to the queue's push and pop operations? Elements are appended to the back of the underlying linked list and removed from the front.
   A) Worst-case push: $\Theta(1)$, Worst-case pop: $\Theta(1)$, Amortized push: $\Theta(1)$, Amortized pop: $\Theta(1)$
   B) Worst-case push: $\Theta(n)$, Worst-case pop: $\Theta(n)$, Amortized push: $\Theta(1)$, Amortized pop: $\Theta(1)$
   C) Worst-case push: $\Theta(n)$, Worst-case pop: $\Theta(1)$, Amortized push: $\Theta(1)$, Amortized pop: $\Theta(1)$
   D) Worst-case push: $\Theta(1)$, Worst-case pop: $\Theta(n)$, Amortized push: $\Theta(1)$, Amortized pop: $\Theta(1)$
   E) Worst-case push: $\Theta(n)$, Worst-case pop: $\Theta(1)$, Amortized push: $\Theta(n)$, Amortized pop: $\Theta(1)$

7. An *ordered stack* is a stack where the elements always appear in increasing order. This special stack supports the following operations:
   - `.pop()`: removes and returns the top element from the ordered stack
   - `.push(x)`: continually removes elements greater than x from the top of the stack until x is the largest element on the stack; after all removals are complete, x is pushed to the top of the stack

   The following example illustrates a call to `.push(6)` on an ordered stack with elements 1, 3, 4, 5, 7, and 9. To reestablish order, elements 9 and 7 are removed from the top of the stack before 6 is pushed in. The elements 7 and 9 are *not* readded once the push is complete. The stack is implemented using a doubly-linked list, with a pointer to the top element.



   (a) What is the worst-case time complexities of `.pop()` and `.push()` in terms of the stack size n?
   (b) What is the amortized time complexity of `.pop()` and `.push()` in terms of the stack size n?

Chapter 12 Exercise Solutions

1. **The correct answer is (C).** Statement I is false because amortized complexity averages out the cost of a single operation within a sequence of operations, which is at least as good as the worst-case time complexity of the operation itself. Statement II is false because amortized analysis is most useful when the worst-case time complexity is a lot worse than the average performance of the operation, but it happens rarely. Statement III is true, as amortized analysis gives you a better idea of what the true upper bound of a sequence of operations is if individual calls to each operation may have different costs.

2. **The correct answer is (B).** In amortized analysis, we average the time required to perform a sequence of operations over all the operations performed. Since the special operation `.shove_back()` works similarly to `.push_back()` when the array is not at capacity, we would expect such a call for `.shove_back()` to take $\Theta(1)$ time. However, when the array is at full capacity, we have to reallocate and sort the contents of the original array, which takes a dominating $\Theta(n\log(n))$ time. Using aggregate analysis, we therefore see that the amortized complexity of a call to `.shove_back()` is $\Theta(n\log(n))/n = \Theta(\log(n))$.

3. **The correct answer is (C).** Increasing the capacity multiplier by a different constant (in this case, 1.5 instead of 2) does not change the fact that the total work performed over n operations is $\Theta(n)$ (you can see this using the sum of a geometric series). As a result, the amortized time complexity remains $\Theta(n)/n = \Theta(1)$.

4. **The correct answer is (A).** Increasing the capacity by a fixed constant results in a $\Theta(n^2)$ total amount of work (as demonstrated using the example at the end of section 12.2). As a result, the amortized time complexity becomes $\Theta(n^2)/n = \Theta(n)$, which is worse than the $\Theta(1)$ amortized complexity that comes with doubling capacity during reallocation.

5. **The correct answer is (A).** The total amount of work required to perform n operations can be expressed as:

   $$T(n) = 1+2+1+4+1+1+1+8+1+\ldots = (1+1+1+\ldots)+(1+2+4+8+\ldots) < n+\Theta(n) = \Theta(n)$$

   Using aggregate analysis, we conclude that the amortized time complexity of a single operation is $\Theta(n)/n = \Theta(1)$. (Note: the scenario described in this problem is exactly what happens during vector reallocation.)

6. **The correct answer is (A).** Unlike vectors, lists do not perform any reallocation when elements are inserted, and the doubly-linked nature of the `std::list<>` ensures that the worst-case time complexity of pushing and popping is $\Theta(1)$. It follows from this that the amortized time complexities of these operations must also be $\Theta(1)$, since running a sequence of n pushes and pops will take at most $\Theta(n)$ time in total, resulting in an amortized complexity of $\Theta(n)/n = \Theta(1)$.

7. (a) In the worst case, `.pop()` takes $\Theta(1)$ time since it is guaranteed to remove only one element from the stack, but `.push()` takes $\Theta(n)$ time, since you may have to remove all the other elements in the stack if you push in a value that is lower than everything in the stack.

   (b) If we look at the lifetime of any element that enters and leaves the stack, the total cost associated with that element is at most 2 (1 to push into the stack, and 1 when it is popped out). Thus, for any sequence of n pushes and pops, the total cost incurred cannot be greater than $2n$. Thus, the amortized complexity of push and pop are $2n/n = \Theta(1)$ (essentially, a more expensive push also gets rid of more elements from the stack, which makes future pushes less likely to be as costly, resulting in a constant amortized cost for each operation). You can look at the queue with two stacks problem in section 12.5 for a similar example.