Assignment 1 Part 1 – Amortized Analysis

Spring 2026

Due: Thursday, Feb 12 11:59 pm

To-do

Answer the following questions in detail. Your answer should be well structured and clearly explained. Feel free to cite references where applicable.

1. Use aggregate analysis to determine the amortized cost per operation for a sequence of n operations on a data structure in which the i^{th} operation costs i if i is an exact power of 2, and 1 otherwise.

Operation Number (i)	1	2	3	4	5	6	7	8	9
Cost	1	2	1	4	1	1	1	8	1
Σcosts for i operations	1	3	4	8	9	10	11	19	20

For calculating cost using the aggregate method, we take the sum of costs and attempt to equally distribute them over all the operations in the sequence. Thus for n operations, the aggregated cost of each operation is (Σcosts)/n. Since the sum of costs for i operations is recursively defined, this cannot be written in a simpler way without being redundant.

2. You perform a sequence of PUSH and POP operations on a stack whose size never exceeds k. After every k operations, a copy of the entire stack is made automatically, for backup purposes. Show that the cost of n stack operations, including copying the stack, is $O(n)$ by assigning suitable amortized costs to the various stack operations. [Use the accounting method]

For use of the accounting method here our expensive operation is considered to be the copying of the stack every k operations. Consider:

- The kth operation requires a cost that cannot exceed k+1. Through the accounting method building credit on each push operation, this cost is essentially reduced to 0.
- Push operations would usually have a cost of 1, but we'll assign push's cost to be 2 such that we build ĉ (credit) > c (actual cost).
- Pop operations essentially 'refund' the cost of the popped element, since it won't have to be copied following the kth operation.

Thus, by the kth operation we have built credit equal to the number of elements in the stack and get the copied stack "for free". Thus for n stack operations the total cost is O(n) as each operation has an amortized cost of O(1).

3. Consider an ordinary binary min-heap data structure supporting the instructions INSERT and EXTRACT-MIN that, when there are n items in the heap, implements each operation in $O(log\ n)$ worst-case time. Give a potential function φ such that the

amortized cost of i NSERT is $O(log\ n)$ and the amortized cost of EXTRACT-MIN is $O(1)$, and show that your potential function yields these amortized time bounds. Note that in the analysis, n is the number of items currently in the heap, and you do not know a bound on the maximum number of items that can ever be stored in the heap.

To start by reiterating the above, the insert and extract-min operations run in a worst case O(log n) time. Thus for n operations the actual cost would be (n log n). To show that the amortized cost of extract-min is O(1), we'll build potential on all of our insert operations. If we say that for each insertion we pay a cost of log n and build potential log n, we get that φ = n log n and ĉ = log n + Δ φ. Since ĉ for an insertion now equals 2 log n its amortized complexity is still O(log n) and the potential means that for any extract-min operation, its cost has already been paid for and its amortized cost is O(1).

4. In a priority queue implemented using a binary heap, some operations (like insertions and deletions) might take longer when the heap needs to be restructured. Explain, in words, how amortized analysis is used to show that the average cost per operation (insert, delete) over a sequence of operations remains efficient, despite occasional restructuring of the heap.

The goal of amortized analysis is to equalize cost over all of the operations made on a data structure to give a more complete view on its efficiency. While the cost of individual inserts and deletes might be greater when restructuring is required, amortized analysis shows that that cost is balanced out by the cheaper operations that don't require restructuring.

5. Show the amortized analysis of Binary Counter implantation from our class slides using aggregate, accounting, and potential methods.

Aggregate:

Value	A[4]	A[3]	A[1]	A[0]
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0

Since the complexity is represented by the number of bits flipped, we can define the total cost for n operations as a sequence of how often A[i] is flipped. A[0] is

flipped n times, A[1] is flipped n/2 times and so on. Thus the total cost for n operations can be represented as 2n. Thus the total amortized cost of n operations is O(n).

Accounting:

We want to build credit on bits that are 1 so that the more expensive operation of flipping many 1s to 0s becomes free. To achieve this, we make the amortized cost ĉ for an increment be 2, as each increment can only create a maximum of 1 new instance of 1 within the structure. This way when many 1s are flipped to 0s, the cost of that operation has already been paid for and the amortized cost of n operations is O(n).

Potential:

For the potential method we'll build potential on increments that flip few bits. Thus we can define our potential function φ = 2n for n increments. As such, we get that the amortized cost for n operations being ĉ = n + Δ φ = O(n).

6. A splay tree is a self-adjusting binary search tree in which each operation (insert, delete, search) involves a process called "splaying," which moves the accessed node to the root. Describe how amortized analysis helps demonstrate that the amortized time complexity of a series of operations on a splay tree can be bounded by $O(logn)$, even though some individual operations may take longer. Show the proof of this for the ZagZig rotation.

While splay trees have some overhead for the splay operation, amortized analysis helps show that despite this overhead, the splaying operation improves the optimization of splay trees. We can define our potential function as being equal to the sum of ranks for the whole structure, where the rank of a node is defined by r(x) = log s where s is the subtree size.

Now for our 2 rotations we'll define the grandparent g, parent p, and child x as the nodes to be rotated. We can now define amortized cost $\hat{c} = c + \Delta \varphi$. Since the actual cost is 2 (for a zag and a zig rotation) we can write $\hat{c} = 2 + r'(x) + r'(p) + r'(g) – r(x) – r(p) – r(g)$. This works because the potential functions measures ranks, and $\Delta \varphi$ measures a change in sum of ranks for g,p, and x.

Since the final position of x is the same as the initial position of g we can simplify this to $\hat{c} = 2 + r'(p) + r'(g) – r(x) – r(p)$. We can rewrite an upper bound of this as $\hat{c} <= 2 + r'(x) + r'(g) – 2r(x)$.

Since $r'(g) <= 2r'(x) – r(x) – 2$, we can substitute this into the above inequality to show $\hat{c} <= 2 + r'(x) + 2r'(x) – r(x) – 2 – 2r(x)$ which simplifies to $\hat{c} <= 3(r'(x) – r(x))$.

Replacing the rank function with its definition and ignoring the coefficient of 3 shows that $\hat{c} <= \log s'(x) – \log s(x)$ where properties of logs with like bases means it can be rewritten as $\hat{c} <= \log(s'(x) / s(x))$ Since x is slayed to the root, $s'(x) = n$ and $s(x) = 1$.

Thus ĉ <= log(n / 1) Thus the amortized time complexity of a series of operations can be bounded by O(log n).

Total Points (60)

- Each question is worth 10 points

Deliverables:

- A pdf file named A1_part_1 uploaded to D2L Dropbox and GitHub