CPSC 320: Amortized Analysis Solutions[*]

As mentioned in the readings, *amortized analysis* is a technique that can be used to obtain a tight upper bound on the worst-case running time of a sequence of operations. It is used in cases where the running time of individual operations may vary widely, and where assuming every operation takes its worst-case running time would not give a good upper-bound. In this worksheet, we will look at two examples where amortized analysis is useful: the first one is simple, but rather useless in practice. However it will help you gain a firmer understanding of how amortized analysis works. The second example is much more practical, but also slightly more complicated.

1 Implementing a queue using two stacks

How often have you found yourself needing a queue, but stuck using a programming language where the standard library only provides a stack data structure? Well, ok. Probably never. However it **is** possible to implement a queue using a pair of stacks, and it's a great example where amortized analysis is necessary in order to get a tight bound on the worst-case running time of a sequence of n operations (even if it's a silly example for pretty much any other purpose).

1. List at least 3 examples of a small sequence of operations (6 to 12 operations) on a queue, with different interleavings of calls to `enqueue` and `dequeue`.

   SOLUTION: Here is the first example: we have a lot of calls to `enqueue`, followed by many calls to `dequeue`.

   ```
   enqueue(4), enqueue(9), enqueue(1), enqueue(17), enqueue(5), dequeue(), dequeue(),
   dequeue(), dequeue()
   ```

   Here is another example, where each call to `enqueue` is immediately followed by a call to `dequeue`.

   ```
   enqueue(4), dequeue(), enqueue(9), dequeue(), enqueue(1), dequeue(), enqueue(17),
   dequeue(), enqueue(5), dequeue()
   ```

   Finally, here is an example where both of the behaviours from the previous two examples occur:

   ```
   enqueue(4), enqueue(9), enqueue(11), dequeue(), dequeue(), enqueue(17), enqueue(5),
   dequeue(), enqueue(0), dequeue()
   ```

2. Show how to implement the `isempty`, `enqueue` and `dequeue` operations on a `queue` data structure that is implemented using two stacks. You can only manipulate the elements of your queue data structure by using `push`, `pop` and `isempty` operations on the two stacks.

   Assume that `pop` returns the element that is popped from the stack, and that `dequeue` should behave similarly for your queue implementation. The `isempty` function returns `True` if the data structure is empty, and `False` otherwise. We provide the constructor below:

   ```
   Queue():
       stack1 = new Stack()
       stack2 = new Stack()
   ```

[*]Copyright Notice: UBC retains the rights to this document. You may not distribute this document without permission.

SOLUTION:

```
isempty():
    return stack1.isempty() and stack2.isempty()

enqueue(x):
    stack1.push(x)

dequeue():
    if stack2.isempty()
        while (not stack1.isempty())
            stack2.push(stack1.pop())

    if stack2.isempty()
        throw exception "Can not dequeue from an empty queue."

    return stack2.pop()
```

3. Define a potential function $\Phi(Q)$ for the queue Q in your implementation. Hint: think about the amount of work that still remains to be done for the elements in the data structure.

   SOLUTION: There are two different potential functions that could be used, depending on what work you believe remains to be done once an element has been enqueued. If you believe the element will need to be moved from the first stack to the second stack (one operation) and then dequeued (another operation) then you want to define $\Phi(Q)$ by

   $\Phi(Q) =$ twice the size of `stack1` + the size of `stack2`.

   If on the other hand you believe that moving an element from one stack to the other requires two operations (a `pop` followed by a `push`) then you will want to define $\Phi(Q)$ by

   $\Phi(Q) =$ three times the size of `stack1` + the size of `stack2`.

4. Compute the *amortized cost* of an `isempty` operation by adding its real cost to the potential difference that results from the `isempty` operation.

   SOLUTION: The real cost of an `isempty` operation is constant (we will call that constant 2) since it's two calls to the stack operation `isempty`. The `isempty` operation does not change the potential of Q. Hence the amortized cost of `isempty` is

$$cost_{am}(\texttt{isempty}) = 2 + 0 = 2$$

5. Compute the *amortized cost* of an `enqueue` operation by adding its real cost to the potential difference that results from the `enqueue` operation.

   SOLUTION: In this solution, we assume we defined $\Phi(Q)$ by

   $\Phi(Q) =$ three times the size of `stack1` + the size of `stack2`.

   The real cost of an `enqueue` operation is constant (we will call that constant 1) since it's a single call to the stack operation `push`. The potential of Q increases by 3 as a result of the `enqueue`, because we are adding one element to `stack1`, so the amortized cost of `enqueue` is

$$cost_{am}(\texttt{enqueue}) = 1 + 3 = 4.$$

6. Compute the *amortized cost* of a `dequeue` operation by adding its real cost to the potential difference that results from the `dequeue` operation.

   SOLUTION: We once again assume that we defined $\Phi(Q)$ by

   $\Phi(Q) =$ three times the size of `stack1` $+$ the size of `stack2`.

   The real cost of the `dequeue` operation depends on the number of elements that need to be transferred from stack `stack1` to stack `stack2`. Let k be that number; if `stack2` is not empty then $k = 0$. The real cost of `dequeue` is $2k + 1$ ($2k$ to move k elements from `stack1` to stack `stack2`, and 1 to `pop` the element being dequeued from `stack2`).

   The potential of Q first decreases by $2k$ when we transfer the k elements from `stack1` to `stack2`, and then by 1 more when we `pop stack2`. So the amortized cost of `dequeue` is

   $$cost_{am}(\texttt{dequeue}) = (2k + 1) - (2k + 1) = 0$$

7. What is the worst-case running time of a sequence of n operations on an initially empty queue?

   SOLUTION: The worst-case running time of a sequence of n operations on an initially empty queue is bounded above by

   $$\sum_{i=1}^{n} cost_{am}(op_i)$$

   where op_i is the i^{th} operation. Based on the previous three steps $cost_{am}(op_i) \leq 4$. Therefore the worst case running time of a sequence on n operations on our data structure is at most $4n$, which is in $O(n)$, even though a single `dequeue` operation real cost might be in $\Omega(n)$ (see the first call to `dequeue` in our first small example).

2 Hash Tables that grow and shrink[1]

Suppose that we want to implement a dynamic, open-address hash table. Recall that the load factor α is defined by:

$$\alpha = n/s$$

where n is the number of elements in the hash table, and s is the size of the array used to store these elements. One possible scheme to make sure that the array is neither too big nor to small is the following:

- If an insertion causes α to become greater than or equal to 90%, then we allocate a new array whose size is 1.5 times the size of the old array, and reinsert all of the elements into the new, larger array.
- If a deletion causes α to become less than or equal to 40%, then we allocate a new array whose size is 2/3rd the size of the old array, and reinsert all of the elements into the new, smaller array.

In this section of the worksheet, we use amortized analysis to prove that every sequence of t operations on an initially empty hash table using this scheme runs in $O(t)$ time. We will assume the following:

- inserting or deleting an element (without reallocating the array) takes $\Theta(1)$ time.
- creating a new array A' to replace an old array A, and copying all of the elements of A into A', takes time in $\Theta(\max\{\text{length}(A), \text{length}(A')\})$. For simplicity, we will further assume the multiplicative constant hidden by the Θ notation is 1.

1. How full will the array be after it is reallocated because it became 90% full?

   SOLUTION: If the array becomes 90% full, then after it is reallocated it is only $90\%/1.5 = 60\%$ full.

2. How full will the array be after it is reallocated because it became only 40% full?

   SOLUTION: If the array becomes only 40% full, then after the reallocation it is $40\% * 1.5 = 60\%$ full.

3. We now want to define a potential function for the data structure. Unlike the previous example and those from the readings, the potential function will not depend on the number of elements in the hash table. What form will this potential function take? Hint: think about the fact that reallocating the array can not happen too frequently.

   SOLUTION: Suppose we have just reallocated the array. Because 60% is not very close to either 40% or 90%, a minimum number of operations will need to be performed before the array has to be reallocated again. We want these operation to generate potential that we will then be able to use to "pay" for the cost of the reallocation. So the potential of our data structure will be proportional to the number of operations that were performed on the hash table since the last reallocation. That is, we want $\Phi(H)$ to be of the form

   $$\Phi(H) = k \times \text{ the number of operations since the last reallocation}$$

   where k is a positive integer. It turns out that $k = 5$ will work, so we will use

   $$\Phi(H) = 5 \times \text{ the number of operations since the last reallocation}$$

[1]like Alice in Wonderland.

4. What is the amortized cost of an insert operation on the hash table? You must take two cases into account: the case where no reallocation was necessary, and the case where one was.

   SOLUTION: An insertion that does not result in a reallocation has a real cost of 1, and the potential difference is $\Phi(D_i) - \Phi(D_{i-1}) = 5$. So the amortized cost of the operation is 6.

   Now suppose that after the insertion we need to reallocate the array. After the previous reallocation, the array was 60% full (it does not matter if the reallocation was to increase the array's size, or to decrease it). To get from 60% full to 90% full, we needed at least $30\% \cdot s$ insert operations, where s is the size of the array before we perform the reallocation. The real cost of the reallocation is $1.5s$. The potential difference is

$$\Phi(D_i) - \Phi(D_{i-1}) = 0 - 5 * 0.3s = -1.5s$$

   Hence the amortized cost of the reallocation is $1.5s - 1.5s = 0$.

   Therefore the amortized cost of insert is 6, whether or not a reallocation occurred.

5. What is the amortized cost of a delete operation on the hash table? You must take two cases into account: the case where no reallocation was necessary, and the case where one was.

   SOLUTION: A deletion that does not result in a reallocation has a real cost of 1, and the potential difference is $\Phi(D_i) - \Phi(D_{i-1}) = 5$. So the amortized cost of the operation is 6.

   Now suppose that after the deletion we need to reallocate the array. After the previous reallocation, the array was 60% full (it does not matter if the reallocation was to increase the array's size, or to decrease it). To get from 60% full to 40% full, we needed at least $20\% \cdot s$ delete operations where s is the size of the array before we perform the reallocation. The real cost of the reallocation is s. The potential difference is

$$\Phi(D_i) - \Phi(D_{i-1}) = 0 - 5 * 0.2s = -s$$

   Hence the amortized cost of the reallocation is $s - s = 0$.

   Therefore the amortized cost of delete is 6, whether or not a reallocation occurred.

6. What is the worst-case running time of a sequence of t operations on an initially empty hash table?

   SOLUTION: The worst-case running time of a sequence of t operations on an initially empty hash table is bounded above by

$$\sum_{i=1}^{t} cost_{am}(op_i)$$

   where op_i is the i^{th} operation. Based on the previous two steps $cost_{am}(op_i) = 6$. Therefore the worst case running time of a sequence on t operations on our data structure is at most $6t$, which is in $O(t)$, even though a single dequeue operation real cost might need to copy every element currently in the hash table into a new array.