Tutorial: Amortized Analysis and Fibonacci Heaps

---

§1 Amortized Analysis

Problem 1. Suppose that in a k-bit binary counter supporting only increment operations, the counter is not initialised to 0 (so that the first increment can start at any value). Will the $O(1)$ bound the amortized cost of increment operation in a sequence of n increment operations still hold in such a counter ?

Problem 2. Redo the above question if counter is initialised to a value which is of the order $O(n)$.

Problem 3. Redo the above question if counter $= 2^{\Theta(n)}$.

Problem 4. Will the $O(1)$ amortized cost of increments in a k-bit binary number remain valid if you also allow decrement operations (i.e. in any sequence of intermixed increments and decrement operations) ? If yes, show the amortized analysis. If not, justify why not.

Problem 5. A sequence of stack operations is performed on stack whose size never exceeds k. After every k operations the copy of the entire stack is made for backup purposes. Show that the cost of n stack operations including copying the stack is $O(n)$ using the counting method.

Problem 6. Consider the implementation of a Queue using two stacks A and B. Find the amortized cost of a sequence of n enqueue and dequeue operations using each of aggregate, accounting and potential method.

Problem 7. Consider a stack with a fixed size k with PUSH and POP operations (assume PUSH operations fail in $O(1)$ time if stack is full and POP operations fail in $O(1)$ time if stack is empty). Which of the following are NOT valid potential function for amortized analysis of a sequence of n PUSH and POP operations (choose all that apply).

(a) The number of elements in the stack

(b) The number of free spaces in the stack

(c) Number of free spaces - Number of elements in the stack

(d) Number of free spaces + Number of elements in the stack

Problem 8. Suppose we perform 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. Use aggregate analysis or accounting method to determine the amortized cost per operation.

Problem 9 (Dictionary Problem)**.** One of the most common classes of data structures are the **"dictionary" data structures** that support fast **insert** and **lookup** operations into a set of items.
Suppose you implement a dictionary using a sorted array. This would be good for lookups (using binary search we can take up to $O(logn)$ time) but bad for insertions (can take linear time). If you used a linked list, than inserts are efficient ($O(1)$) but the lookups are not (they can take linear time).
Here is a simple array-like data structure that can provide better than (amortized) linear options for both insertions and lookups. The main idea is to use a collection of arrays, where array i has size 2^i . Each array is either empty or full, and each is in sorted order. There is no relationship between items in different arrays. To know which arrays should be full or empty, consider the binary representation of the number of items and use the arrays in which the bit is non-zero. For instance, 11 items (11 = 1+2+8) would use the arrays of sizes 1, 2 and 8, with the other arrays being empty. This would result in something like:

```
A[0] = {6}
A[1] = {2,13}
A[2] = empty
A[3] = {1,6,7,8,10,15,17,29}
```

(a) Suppose that in order to do a lookup we do a binary search in each filled array. How much time will this take in the worst case?

(b) And what about inserts? Suppose we start by creating a new array of size 1 with the new item. We then check to see if $A[0]$ if full. If it is empty, we are done (just put the new array in position 0). If not, then we merge the (sorted) contents of the new array with $A[0]$ and we now verify if $A[1]$ is full, and so on, stopping when we find an array that is not full. Experiment be "manually" inserting some numbers to see what happens. What is now the worst case for an insertion? And what is the amortized cost of an insertion? [hint: use the result of Binary Counter problem]

Problem 10. In this problem we consider a *monotone priority queue* with operations Init, Delete, and DeleteMin. Consider the following implementation using a boolean array A:

```
Init(n)
    for i=1 to n do
        A[i]=true
    end
end

Delete(i)
    A[i]=false
    end

DeleteMin()
    i=1
    While
        A[i]=false do
            i=i+1
        end
        if i=<|A| then
            Delete(i)
            return i
        else
            return 0
        end
    end
```

(a) Analyze the running time of each of the procedures.

(b) Describe a simple modification to DeleteMin such that it has amortized running time $O(1)$ (while maintaining the running times of Init and Delete). Explicitly give the potential function used in your analysis.

(c) Describe a different implementation such that both Delete and DeleteMin have worst-case running time $O(1)$.

Problem 11. An *ordered stack* **S** is a stack where the elements appear in increasing order. It supports the following operations:

- INIT(S): Create an empty ordered stack.
- POP(S): Delete and return the top element from the ordered stack.

- PUSH(S, x): Insert x at top of the ordered stack and re-establish the increasing order by repeatedly removing the element immediately below x until x is the largest element on the stack.
- DESTROY(S): Delete all elements on the ordered stack.

Example

The following shows an example of an ordered stack and the same stack after performing a PUSH(S,2) operation (the order is re-established by removing 7, 5, and 3)

Intial Stack: 7, 5, 3, 0, -3, -5, -6. Top pointer points to 7
Final Stack (after removal): 2, 0, -3, -5, 6. Top pointer points to 2

Like a normal stack we implement an ordered stack as a double linked list (maintaining a pointer to the top element).

(a) What is the worst-case running time of each of the operations INIT, POP, PUSH, and DESTROY?

(b) Argue that the amortized running time of all operations is $O(1)$.

NOTE: Do all Problems of CLRS

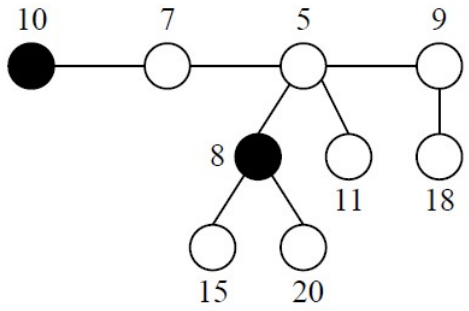
§2 Fibonacci Heaps

Problem 12. Do the following:

(a) Choose 12 distinct numbers x_1, x_2, ..., x_{12} randomly from 1 and 100 (just pick without thinking). In your answer, show first the numbers x_1, x_2, ..., x_{12} clearly in one line, in the format $x_1 = ..$, $x_2 = ..$, and so on.

(b) Perform the following operations on an initially empty Fibonacci heap in this order: Insert x_5, Insert x_3, Insert x_7, Insert x_4, Insert x_8, Insert x_{11}, ExtractMin, Insert x_1, Insert x_9, Insert x_6, Insert x_{10}, Insert x_2, Insert x_{12}, ExtractMin. In any operation, whenever you have a choice between left and right for doing something, choose left. In your answer,
(i) Draw the heap after the first ExtractMin, and
(ii) Draw the heap after the second ExtractMin.
Do not show anything else, and no explanations are needed.

Problem 13. Consider the following Fibonacci Heap (the values stored at each node are shown against each node, and the marked nodes are shown as black). Show the new heap after ExtractMin is called on the heap (just show the new heap, no explanation is needed). Whenever you have a choice of going left or right in the root list, go right.



NOTE: Do all Problems of CLRS